IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Patent Application No.:	09/881,424	) Group Art Unit: 2671
Filing Date:	14 June 2001	)
Inventor:	Wittenbrink	)
Title:	SYSTEM FOR PROCESSING OVERLAPPING DATA	) Examiner: Hadidi, John
		) Atty Docket: 10001077-1

DECLARATION UNDER 37 U.S.C. §131

Assistant Commissioner for Patents
Commissioner of Patents and Trademarks
Washington, D.C. 20231

I, Craig Wittenbrink, hereby declare as follows.

1. All statements set forth herein made on my own personal knowledge are true, and all statements made on information and belief are believed to be true.

2. I am the sole inventor of the patent application identified above.

3. Attached hereto as Exhibit A is a true and correct copy of an invention disclosure document I submitted to the real-party in interest (Hewlett-Packard Co.) on November 11, 1999.

4. Exhibit A as submitted on November 11, 1999, and attached hereto, shows complete conception and reduction to practice of all claims of the patent application identified above, and represents the principle materials used in the preparation of that patent application.

5. As indicated by Exhibit A, I was in complete possession of the invention of all claims of my patent application, and had reduced that invention to practice, prior to July 19, 2000.

I, Craig Wittenbrink, acknowledge that willful false statements and the like are punishable by fine or imprisonment, or both (18 U.S.C. 1001) and may jeopardize the validity of the application or any patent issuing thereof.

Craig M. Wittenbrink — Dec 17, 2004 — Date

H **INVENTION DISCLOSURE** PAGE ONE OF ____

PDNO 1000 1077 DATE RCVD 11-11-99 ATTORNEY MPS

Instructions: *The information contained in this document is* ***COMPANY CONFIDENTIAL*** *and may not be disclosed to others without prior authorization. Submit this disclosure to the HP Legal Department as soon as possible. No patent protection is possible until a patent application is authorized, prepared, and submitted to the Government.*

CSL-VC

Descriptive Title of Invention: True Transparency with The Fragment Buffer Graphics Architecture

Name of Project: graphics

Product Name or Number: Bonanza / fx graphics

Was a description of the invention published, or are you planning to publish? If so, the date(s) and publication(s):

NO

Was a product including the invention announced, offered for sale, sold, or is such activity proposed? If so, the date(s) and location(s):

NO

Was the invention disclosed to anyone outside of HP, or will such disclosure occur? If so, the date(s) and name(s):

NO

If any of the above situations will occur within 3 months, call your IP attorney or the Legal Department now at 1-888-4919 or 970-898-4919.

Was the invention described in a lab book or other record? If so, please identify (lab book #, etc.)

Yes HPL 2373

Was the invention built or tested? If so, the date:

Yes Sep, 21, 1999

Was this invention made under a government contract? If so, the agency and contract number:

NO

Description of Invention: *Please preserve all records of the invention and attach additional pages for the following. Each additional page should be signed and dated by the inventor(s) and witness(es).*

See Attached Addendum and HPL-XX Technical Report

A. Prior solutions and their disadvantages (if available, attach copies of product literature, technical articles, patents, etc.).
B. Problems solved by the invention.
C. Advantages of the invention over what has been done before.
D. Description of the construction and operation of the invention (include appropriate schematic, block, & timing diagrams; drawings; samples; graphs; flowcharts; computer listings; test results; etc.)

Signature of Inventor(s): Pursuant to my (our) employment agreement, I (we) submit this disclosure on this date: [ ].

Employee No.	Name	Signature	Telnet	Mailstop	Entity & Lab Name
491791	Craig M. Wittenbrink	Craig M. Wittenbrink	857-2329	3U-4	HPL CRL

(If more than four inventors, include additional information on another copy of this form and attach to this document)

Form 3.1 IDF.DOC Rev. 10/25/99

H. **INVENTION DISCLOSURE** COMPANY CONFIDENTIAL PAGE ____ OF ____

Signature of Witness(es): *(Please try to obtain the signature of the person(s) to whom invention was first disclosed.)*

The Invention was first explained to, and understood by, me (us) on this date: [ ]

Full Name	Signature	Date of Signature
Kevin Wu	Kevin Wu	Nov. 10, 1999
Full Name	Signature	Date of Signature

Inventor & Home Address Information: *(If more than four inventors, include addl. information on a copy of this form & attach to this document)*

Inventor's Full Name: Craig Michael Wittenbrink

Street: 1132 Madison Avenue

City: Redwood City State: CA Zip: 94061

Do you have a Residential P.O. Address? P.O. BOX City State Zip

Greeted as *(nickname, middle name, etc.)* Citizenship: USA

Inventor's Full Name

Street

City State Zip

Do you have a Residential P.O. Address? P.O. BOX City State Zip

Greeted as *(nickname, middle name, etc.)* Citizenship

Inventor's Full Name

Street

City State Zip

Do you have a Residential P.O. Address? P.O. BOX City State Zip

Greeted as *(nickname, middle name, etc.)* Citizenship

Inventor's Full Name

Street

City State Zip

Do you have a Residential P.O. Address? P.O. BOX City State Zip

Greeted as *(nickname, middle name, etc.)* Citizenship

INVENTION DISCLOSURE ADDENDUM

TO: MARC SCHUYLER

FROM: CRAIG M. WITTENBRINK

SUBJECT: TRUE TRANSPARENCY WITH THE FRAGMENT BUFFER GRAPHICS ARCHITECTURE

DATE: 11/09/99

CC:

DESCRIPTION OF INVENTION

a. **prior solutions and their disadvantages**

Prior solutions include: 1) software reording of graphics primitives at the application level, for example as necessary in OpenGL [OpenGL92] 2) multipass rendering with a dedicated augmented frame buffer [Mammen89]; 3) limiting users to 4 levels with a hardware chip [Kelley94][Winner97]; 4) Software A-buffer renderer [Carpenter84]; 5) screen door transparency such as that implemented in the SGI Reality Engine [Akeley93]; 6) a dedicated sorting network and RAM buffering [Baker94].

The disadvantages of the software techniques (1, 2, 4), are the inefficiency of having an application sort primitives. The sorting is viewpoint dependent, so for 3D graphics rendering, as a viewpoint is altered a new sorting of all primitives would be done, or held in a previously created data structure such as an octree. Such pre-sorting may require cutting primitives that are intersecting other primitives to break potential cycles.

The disadvantages of the previous hardware techniques, are either quality trade-offs, such as only supporting a fixed number of layers (3), reducing the spatial resolution via a dithering technique called screen door transparency, which also supports only a fixed number of transparent layers (5), or tremendous cost with many dedicated memories and circuitry only for the sorting (6).

There has been no prior solution that provides economical true transparency without changing the application, in a graphics 3D rendering architecture.

b. **problems solved by the invention**

The problem solved by the invention is how to compute a 3D graphics rendering of surfaces that are partially transparent, while providing the ease of use of a traditional Z-buffering architecture. The problem is a long standing one in graphics, and involves the proper sorting of the primitives in the depth ordering along view rays. A primary advantage of the invention is how to solve this in hardware, with an economical (few gates) method. Also, there are no trade-offs in quality, as any number of layers at a given pixel may be supported, and the memory cost is amortized across the entire screen.

Craig Wittenbrink

Nov. 10, 1999

c. Description of the construction and operation of the invention

The invention is fully disclosed in the attached HP Confidential technical report [Wittenbrink99]. This invention would be added as described to a 3D rendering graphics ASIC or ASICs, to implement in hardware true transparency. A proposed hardware architecture, as well as specific hardware control are given in the referenced technical report.

d. Advantages of the invention over what has been done before

The advantages of the invention over what has been done before are: the invention does not require modification of the application; the invention does not force the application to sort primitives (triangles) to work properly; the invention is economical, as it requires only simple modifications to the Z-comparison and compositing logic of existing Z-buffering architectures; the invention provides new features without compromising existing features; and the invention may also incorporate antialiasing.

REFERENCES:

[Akeley93] Kurt Akeley. RealityEngine Graphics. In Proceedings of SIGGRAPH, pages 109-116, Anaheim, CA, August 1993. ACM.

[Carpenter84]] Loren Carpenter, The A-buffer, an antialiased hidden surface method. In Proceedings of SIGGRAPH, pages 103-108. ACM, July 1984. Vol. 18, No. 3.

[Baker94] Stephen J. Baker, Dennis A Cowdrey, Graham J. Olive, and Karl J. Wood, Image generator for generating perspective views fromo data defining a model having opaque and translucent features. United States Patent Number 5,363,475, Nov. 8 1994.

[Kelley94] Michael Kelley, Kirk Gould, Brent Pease, Stephani Winner, and Alex Yen, Hardware accelerated rendering of CSG and transparency. In Proceedings of SIGGRAPH, pages 177-184, Orlando, FL, July 1994. ACM.

[Mammen89] Abraham Mammen, Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Computer Graphics and Applilcations, 9(4):43-55, July 1989.

[OpenGL92] OpenGL Architectural Review Board. OpenGL Reference Manual. Addison-Wesley, Reading, MA 1992.

[Winner97] Stephanie Winner, Michael Kelley, Brent Pease, and Alex Yen. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In Proceedings of SIGGRAPH, pages 307-316, Los Angeles, CA, August 1997. ACM.

[Wittenbrink98] Craig M. Wittenbrink, True Transparency with the Fragment Buffer Graphics Architecture. HP Labs Confidential Technical Report, HPL-99-TBD, Nov. 1999.

Craig M. Wittenbrink
Nov. 10, 1999

True Transparency with the Fragment Buffer Graphics Architecture

Craig M. Wittenbrink
Hewlett-Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304

November 10, 1999

The fragment buffer is a new method for providing computation for true transparency of rendered fragments. True transparency is provided without altering the application, without requiring the application to sort the data, and without the deficiencies of previous methods such as screen door transparency and the A-Minus buffer. The fragment queue or fragment buffer can compute true transparency with any number of layers. A variant of the fragment buffer that was designed for minimal hardware complexity, with maximum algorithmic improvement is simulated. Statistics are shown for a variety of different scenes using a trace based methodology, and an instrumented Mesa(TM) OpenGL implementation. The fragment buffer is shown to require from 2.1 to 3.6 times more memory than traditional Z-buffering to provide true transparency. Detailed hardware design is provided, including the state transition diagrams, next state table, and architectural schematics. The fragment buffer can also be used for antialiasing, and an example of Carpenter's classical A-buffer antialiasing is shown. A key invention of antialiasing is to modify Carpenter's recursive algorithm into an iterative front-to-back processing.

1 Introduction

The Fragment Buffer is about achieving greater graphics visual realism through novel use of resources. This paper discusses the architecture and shows examples of a rendering simulator that uses the fragment buffer. Sutherland, Sproull and Schumacker [11] explained hidden surface algorithms as sorting to determine what is visible on the screen. Object space primitives are sorted to screen space locations in X, Y, and Z. Most architectures compare the Z location with the existing values in a Z buffer, and decide what can be thrown out, or overwritten into the Z buffer. The technique is simple, and fast in hardware. It has dominated graphics architectures for nearly two decades. But, there are deficiencies. It is difficult to use Z buffering with complex shading and/or texturing algorithms. The reason is because a large number of pixel values are overwritten, so expensive shading, can be overly burdensome. Another primary difficulty is that Z buffering is a read modify write, and so an actual sort is not being done. Therefore, true transparency is not possible efficiently on a Z buffering architecture. An additional difficulty of Z buffering is that antialiasing is expensive and requires expanding the Z buffer to the number of subsamples used for antialiasing.

Compute intensive shading can be done with a sort last approach [7]. If the sorting of an object primitive to the screen, is left until the last step of the graphics pipeline, it is called a sort last technique from Molnar et al.'s proposed parallel rendering taxonomy. PixelFlow [8] used sort last so that all pixels were determined to be visible or not, and then shaded, which makes shading and/or texturing work proportional to the frame buffer size, not the object complexity. As models become large, this is an important advantage. PixelFlow's primary difficulty is the bandwidth and subdivision necessary to composite entire screens between different graphics pipelines. A sort last approach solves the inefficiencies of the Z buffering architecture, but does not

Craig Wittenbrink
Nov. 10, 1999

provide correct transparency.

Improved methods for correct transparency have been investigated by Mammen [6], Carpenter [3], Kelley et al. [5, 12], and Baker et al. [2]. The proposed techniques are either software only [3], require multiple passes of rendering the geometry [5, 12, 6], and/or only render a fixed number of transparent levels correctly [5, 12]. Transparency is challenging problem to solve in hardware, and other techniques such as screen door, or sorting the polygons back-to-front in the application have been used. There are quality problems with screen door transparency [1], essentially a dithering technique, and requiring applications to send the polygons in sorted order is not general to legacy applications or easy to do.

Improved methods for antialiasing have been investigated throughout the graphics literature, and numerous techniques exist. In hardware, antialiasing has been done most directly through supersampling [1, 8]. Adaptations of the A-buffer [3] to hardware have also been investigated, with partial implementations due to the generality of the A-buffer approach [12]. The difficulty to economically implement antialiasing, has meant that most graphics architectures support antialiasing with multiple passes through the geometry.

Figure 1 on the bottom row shows what happens in OpenGL, when rendering 3 transparent squares of red, green, and blue. A different image results from each different drawing order, even though the 3 squares have a fixed Z depth location. On the top row of Figure 1, different drawing order does not impact the visual appearance. This shows the results of true transparency, with the invention of the fragment buffer.

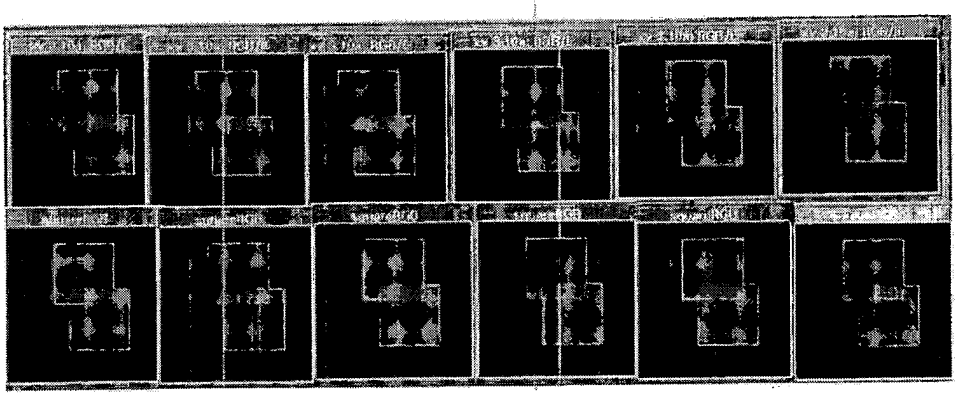


Figure 1: Top row, fragment buffer, same appearance. From near to far, the squares are ordered Blue, Green, Red. Bottom row, OpenGL, different every time.

By the addition of a memory, called the *fragment buffer*, proper transparency ordering and antialiasing can be economically implemented in hardware. I show how, for example, correct transparency can be implemented in such an architecture. I also show how the A-buffer algorithm and adaptive antialiasing or nonuniform sampling can be implemented. Essentially, a frame buffer is used for storing the closest opaque fragment, or the furthest transparent fragment if there aren't any opaque fragments. For pixels with additional fragments, those fragments are sent to the buffer along with their X and Y location. In successive passes, the fragments are considered, and composited [9] (Porter and Duff) into the frame buffer. Only 1 pass is needed for processing the geometry, so no extra storage is needed for geometry, and a single fragment buffer is shared for the entire screen. This amortization of the extra storage over the entire screen allows unique savings over techniques with large per pixel dedicated storage. For architectures that do screen based subdivision, such a fragment buffer fits in naturally. Bucketization of primitives and/or fragments reduces the storage requirements of the X and Y location, as it is addressed only within a tile.

True transparency and adaptive antialiasing have only been attempted in hardware in high end graphics image generators for flight simulation [2]. The use of a fragment buffer is flexible and efficient for the calculation of proper transparency, without multiple passes of the geometry. Multiple passes of fragments are efficiently done, and many fragments are culled, and eliminated with Z and occlusion testing. Screen

Craig Kolb Nov. 09, 1999

space subdivision, which allows for the reduced communication requirements, also provides for reducing the amount of sorting necessary. And, antialiasing can be done efficiently without dedicating a large amount of memory per pixel. The buffer area may be partitioned to provide efficient separation of different types of fragments, and the novel ease of nonuniform sampling in a hardware pipeline. Experiments show the required memory to support true transparency with highly detailed models to be from 2.1 to 3.6 times more memory than a traditional Z buffer. Additionally, a reformulation of Carpenter's software recursive antialiasing technique, shows how to perform iterative front-to-back antialiasing with this proposed hardware. Many issues need to be investigated, such as the implementation of stencils, and the support of all OpenGL modes. But, the advantages, and the proposed performance levels, make the architecture an advance beyond what has been possible.

The main inventions describe how to achieve correct ordering for transparency and how to achieve antialiasing economically. Section 2 describes in the fragment buffer architecture in the context of a Z-buffering graphics rendering architecture. Section 3 shows the state transition diagrams, and next state tables, for the comparison logic. Section 4 shows the results with several test data sets. Section 5 discusses fragment buffer variants, and Section 6 concludes the paper.

2 Fragment Buffer Graphics Hardware

To explore the most economical hardware, an invention that augments a conventional Z-buffer is explored. Many variations of the invention are possible, and are discussed further in Section 5. The scenario is as follows, the architecture is a standard graphics pipeline, with geometry processing (G), and rasterization (R). We add a fragment buffer, and a 2nd Z storage to the frame buffer. This configuration provides the maximum advantage with the minimal additional memory. Figure 2 shows the architecture. Figure 3 shows

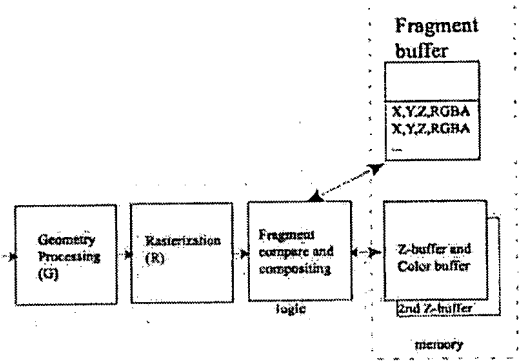


Figure 2: Graphics architecture schematic plus added fragment buffer, and second Z buffer.

more details of the new processing. A fragment coming from rasterization or from the fragment buffer is multiplexed into the fragment buffer comparison and controller. The fragment buffer is considered to be a circular queue, and if the queue overflows because of excessive fragments then it can be paged to systems memory. Because accesses are sequential and used on a first-in first-out (FIFO) basis, performance will degrade gracefully. A fragment is defined as a point sample with color, opacity, and depth resulting from the rasterization (R), as in RGBAZ. Depending on the fragment's opacity, depth, and the previous state of the frame buffer at that location, a fragment may be stored on the fragment buffer, composited into the frame buffer, or discarded. The fragment buffer comparison and controller is where the z-buffering comparison typically takes place. The z-buffering is now augmented and revised to provide true transparency and or antialiasing. The processing is demonstrated through an example.

Nov. 10, 1999

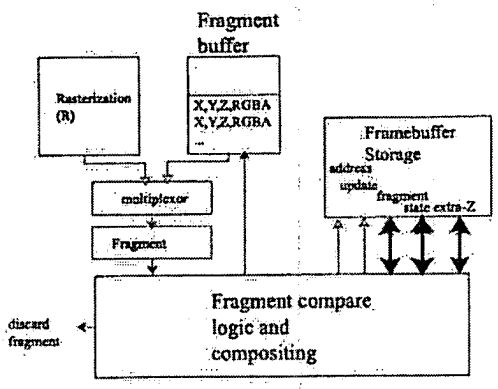


Figure 3: Detailed diagram of interface to fragment buffer and frame buffer.

Figure 4 shows a pixel with 8 fragments, one of which is opaque, O. The transparent fragments are labelled T1 to T4 and Tx, Ty, and Tz. The fragments are drawn in the order shown 1, 2, 3, ..., 8, and processing occurs as shown in the figure. Processing occurs by first considering the fragments during rasterization. This is phase 1. Then, if there have been fragments placed into the fragment buffer, following passes are performed. Only a single pixel per screen location is saved, so when multiple fragments that are unoccluded lie upon a single pixel, they are sent to the fragment buffer. This example has 6 passes: phase 1, phase 2, phase 31, phase 32, phase 33, and phase 34. For a shorthand notation, the frame buffer is labelled B, the next Z value, or 2nd Z value is B_{nz}, next Z prime is B'_{nz}, and the fragment under consideration is F.

Figure 4 shows how during phase 1, any opaque layers are found. In this case transparent fragments, closer than the opaque layer were all queued, T1, T2, T3, and T4. An underline indicates the fragments in the fragment buffer. The transparent layers beyond the opaque layer, Tx and Ty, were queued. In phase 2, these fragments, Tx and Ty, are culled, and the furthest transparent layer's Z is saved. The fragments are processed in the same order each time as they are read from and written to the fragment buffer. Note that the true depth complexity of the pixel is 5, but we took 6 passes. In cases where no further than opaque-transparent-fragments are queued, there is one less pass, 5. Phase 2 culls fragments Ty, and Tx as they are further from the eye point than the opaque layer O. Next, in phase 2, fragment T1, has its Z value saved as B_{nz}, and is put on the fragment buffer, as shown by S. Fragments T2, T3, and T4 are also re-queued. In Phase 31, the frame buffer, B, holds the opaque Z, Oz, and color attributes. B_{nz}, or NextZ, is also in the frame buffer, and holds the proper Z value for the furthest transparent layer, T1. In Phase 31, the fragments come out of the fragment buffer in the same order that they were placed there. First, fragment T1 is read, and it's Z value matches BnZ. Therefore, it is immediately composited with the frame buffer. The next fragment is read, T2, and it is the furthest Z that is closer than the B_{nz}, so the $B'_{nz} = T_{z2}$ (Bnext Zprime) is written, and the fragment is re-queued (S). Fragments T3 and T4 are considered and re-queued (S).

Phase 32 continues the same as phase 31, with the remaining fragments on the fragment buffer. There are T2, T3, and T4. Note that the B'_{nz} or Nextzprime of phase31 is B_{nz}, or Bnextz of phase 32. This alternation of the interpretation of B_{nz} and B'_{nz} continues for each even and odd phase 3 needed. The z storage used is the frame buffer Z, and the 2nd frame buffer Z. Always just 2 Z values per pixel.

Phase 33 starts with a buffer, B, that is the previously composited opaque, and furthest two transparent fragments. B_{nz} is the z value value of fragment T3. T3 is the first fragment considered, so it is matched with B_{nz} and also composited. Fragment, T4's Z is set to B'_{nz}. Then in phase 34, fragment T4 is considered again, matches B_{nz}, and is composited into the final correct pixel color. Once the fragment buffer is emptied processing completes for that frame. The fragment buffer must contain the location of the fragments as fragments for the whole screen are intermixed on the fragment buffer. Figure 5 shows possible fragments

Nov. 19 1999

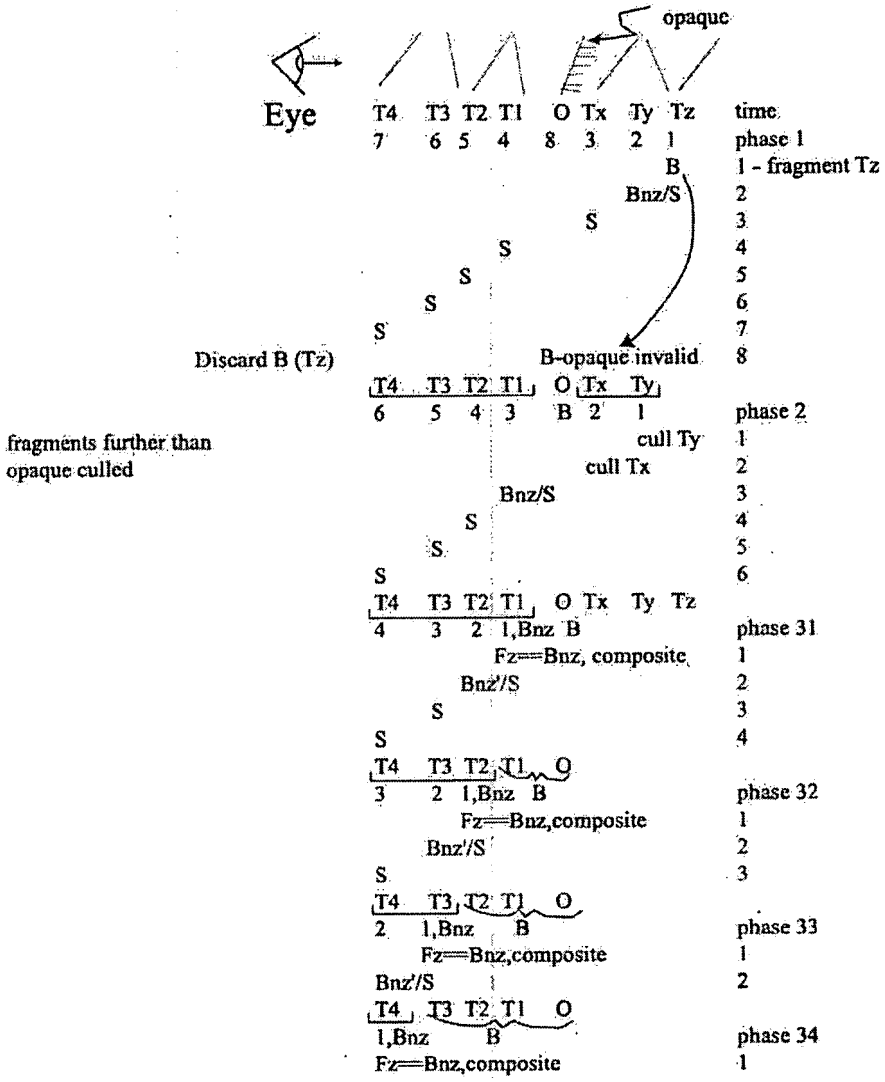


Figure 4: Fragment processing example, for Z-buffer, with Extra Z, and fragment buffer.

physical location	phase 1	phase 2	phase 3 odd	phase 3 even	phase 3 odd
B_{nz}	B_{nz}	B_{nz}	B'_{nz}	B_{nz}	B'_{nz}
B_z	B_z	B'_{nz}	B_{nz}	B'_{nz}	B_{nz}

Table 1: Use of 2nd or extra z buffer, the phase next z prime (Nbz') and Bnextz (B_{nz}) alternate interpretation, while their location is in B_{nz}, or Bz of the frame buffer physical location. See on the left.

Nov 10, 1999

on the fragment buffer, with location (X_o, Y_o) intermixed with (X_p, Y_p). The state machines that control processing for the fragment buffer are described in the next section.

```
...
Xp, Yp, Z2, A2
Xp, Yp, Z4, A4
Xo, Yo, X , X  <=  fragment from a different location
Xp, Yp, Z3, A3
Xp, Yp, Z5, A5
...
```

Figure 5: Fragments on the fragment buffer.

3 State Machine Specification for Single Frame Buffer Solution

Figures 6, 7, and 8 show the state transition diagrams of frame buffer pixels during processing. Figure 9 shows the fragment buffer comparison and controller state beginning with initialization into phase 1 after a new frame. The number of phases varies depending on the frame's depth complexity, as illustrated in the example in the previous section. Phase 1 is always first, and all fragments are considered during rasterization. After all fragments from rasterization have been processed, any fragments that were placed on the fragment buffer are considered in phase 2, and so on. The processing terminates when the fragment buffer is emptied. For phase 3, processing alternates between odd and even phases as shown in Figure 9.

Each frame buffer location has a unique state. So each pixel is a state machine, only, you just need to consider the state machine for the fragment location. The state machine requires 6 states for phase1, 3 states for phase 2, and 3 states for phase 3. The 6 states have been labelled as shown in Table 2.

BOTH_INVALID	initial state, no fragments seen here
VALID_OPAQUE	1 opaque fragment seen and stored
VALID_TRANS	1 transparent fragment seen and stored
OPAQUE_INV	an opaque fragment stored closer than queued fragments
BOTH_VALID_T_O	at least two fragments, opaque and transparent
BOTH_VALID_T_T	at least two fragments, both transparent

Table 2: State assignment definitions for the 6 states in phase 1.

The same state assignments are reused for all 3 phases. Of course interpretation varies between phases. The 2nd and 3rd phases will generate an error, if a fragment is located where zero or 1 were seen in phase 1. Phase 1 states have been separated into 3 columns. In the left column, no fragment has been seen, and therefore B and B_{nz} are both invalid. In the middle column, B is filled. In the right column B and B_{nz} are filled and at least 1 fragment is on the fragment buffer.

For this implementation, the rules for equal valued depths are that the earlier fragment is in back of the following fragments. For a transparent fragment to be seen, it must be less than the opaque Z. After the fragment buffer is emptied, the rendering and compositing are complete. If there are not more than 1 visible/opaque or transparent fragment per pixel, then processing completes in phase 1. With only 2

Craig Kolb
Nov. 10, 1999

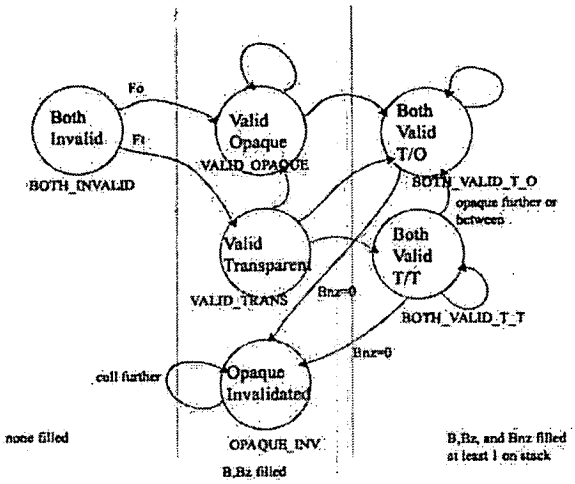


Figure 6: Phase 1 frame buffer pixel state transition diagram.

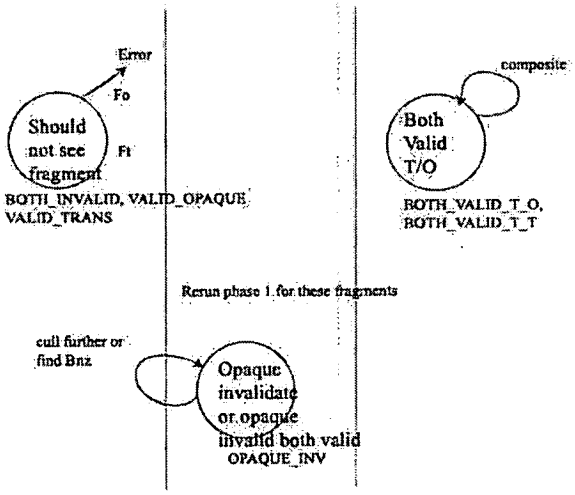


Figure 7: Phase 2 frame buffer pixel state transition diagram.

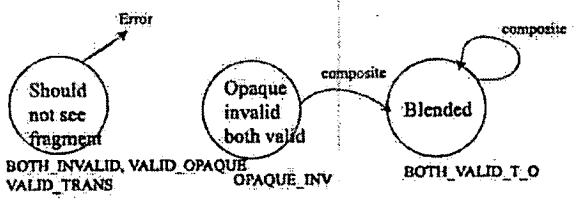
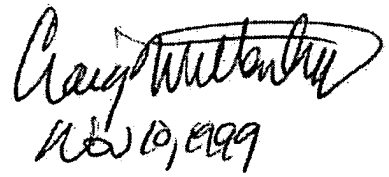


Figure 8: Phase 3 frame buffer pixel state transition diagram. For phase 3 Bnz and Bnz' alternate in storage location between even and odd phases.

Craig Kolb
Nov 10, 1999

current state	inputs	outputs/side-effects	next state
BOTH_INVALID	F_o	$B = F$	VALID_OPAQUE
BOTH_INVALID	F_t	$B = F$	VALID_TRANS
VALID_OPAQUE	$F_z >= B_z$	none (cull fragment)	VALID_OPAQUE
VALID_OPAQUE	F_o, $F_z < B_z$	$B = F$	VALID_OPAQUE
VALID_OPAQUE	F_t, $F_z < B_z$	$B_{nz} = F_z$, queue(F)	BOTH_VALID_T_O
VALID_TRANS	F_o, $F_z <= B_z$	$B = F$, (cull B replace frame)	VALID_OPAQUE
VALID_TRANS	F_o, $F_z > B_z$	queue(B),$B_{nz} = B_z$,$B = F$	BOTH_VALID_T_O
VALID_TRANS	F_t, $F_z <= B_z$	$B_{nz} = F_z$, queue(F)	BOTH_VALID_T_T
VALID_TRANS	F_t, $F_z > B_z$	queue(B), $B_{nz} = B_z$,$B = F$	BOTH_VALID_T_T
BOTH_VALID_T_O	(F_z), $F_z >= B_z$	none (cull fragment)	BOTH_VALID_T_O
BOTH_VALID_T_O	F_o, $F_z < B_z \& F_z > B_{nz}$	$B = F$	BOTH_VALID_T_O
BOTH_VALID_T_O	F_o, $F_z < B_z \& F_z <= B_{nz}$	$B = F$,$B_{nz} = 0$(invalidate Bnz)	OPAQUE_INV
BOTH_VALID_T_O	F_t, $F_z < B_z \& F_z > B_{nz}$	$B_{nz} = F_z$,queue(F)	BOTH_VALID_T_O
BOTH_VALID_T_O	F_t, $F_z < B_z \& F_z <= B_{nz}$	queue(F)	BOTH_VALID_T_O
BOTH_VALID_T_T	F_o, $F_z > B_z$	$B_{nz} = B_z$,queue(B),$B = F$	BOTH_VALID_T_O
BOTH_VALID_T_T	F_o, $F_z <= B_z \& F_z > B_{nz}$	$B = F$,(replace frame)	BOTH_VALID_T_O
BOTH_VALID_T_T	F_o, $F_z <= B_z \& F_z <= B_{nz}$	$B = F$,$B_{nz} = 0$(replace frame)	OPAQUE_INV
BOTH_VALID_T_T	F_t, $F_z > B_z$	$B_{nz} = B_z$,queue(B), $B = F$	BOTH_VALID_T_T
BOTH_VALID_T_T	F_t, $F_z <= B_z \& F_z > B_{nz}$	$B_{nz} = F_z$,queue(F)	BOTH_VALID_T_T
BOTH_VALID_T_T	F_t, $F_z <= B_z \& F_z <= B_{nz}$	queue(F)	BOTH_VALID_T_T
OPAQUE_INV	$F_z >= B_z$	none (cull fragment)	OPAQUE_INV
OPAQUE_INV	F_o,$F_z < B_z$	$B = F$	OPAQUE_INV
OPAQUE_INV	F_t,$F_z < B_z$	queue(F)	OPAQUE_INV

Table 3: State transition table for phase1. queue(X) means to place fragment X on fragment buffer or queue. B-frame buffer, F-fragment being considered, F_o- fragment opaque, F_t- fragment transparent, (F_z) opaque/transparent don't care, F_z- fragment depth value, B_z, frame buffer depth value B_{nz}- frame buffer next z depth value.



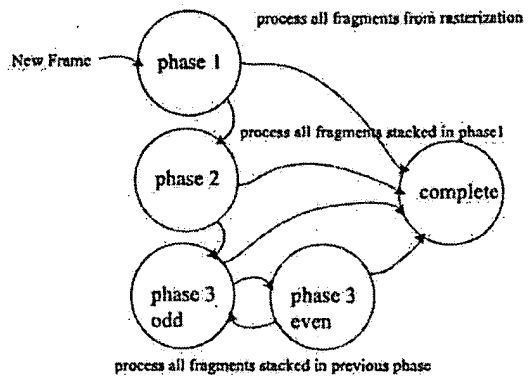


Figure 9: Overall state machine for the Fragment buffer comparison and controller of Figure 3

current state	inputs	outputs/side-effects	next state
BOTH_VALID_T_O	$F_z == B_{nz}$	$B = composite(B, F)$	BOTH_VALID_T_O
BOTH_VALID_T_O	$F_z != B_{nz}, B'_{nz} >= B_{nz}$	$B'_{nz} = F_z$,queue(F)	BOTH_VALID_T_O
BOTH_VALID_T_O	$F_z != B_{nz}, B'_{nz} < B_{nz}, F_z <= B'_{nz}$	queue(F)	BOTH_VALID_T_O
BOTH_VALID_T_O	$F_z != B_{nz}, B'_{nz} < B_{nz}, F_z > B'_{nz}$	$B'_{nz} = F_z$,queue(F)	BOTH_VALID_T_O
OPAQUE_INV	$F_z >= B_z$	none (cull fragment)	OPAQUE_INV
OPAQUE_INV	$F_z < B_z \& F_z > B_{nz}$	$B_{nz} = F_z$,queue(F)	OPAQUE_INV
OPAQUE_INV	$F_z < B_z \& F_z <= B_{nz}$	queue(F)	OPAQUE_INV
(OPAQUE_INV provides same behavior as BOTH_VALID_T_O of phase 1)			

Table 4: State transition table for phase2. queue(X) means to place fragment X on fragment buffer or queue. B-frame buffer, F-fragment being considered, F_o- fragment opaque, F_t- fragment transparent, (F_x) opaque/transparent don't care, F_z- fragment depth value, B_z, frame buffer depth value B_{nz}- frame buffer next z depth value, $composite(B, F)$ Porter and Duff over operator or OpenGL composite.

current state	inputs	outputs/side-effects	next state
BOTH_VALID_T_O	Same as phase 2 above		
OPAQUE_INV	$F_z == B_{nz}$	$B = composite(B, F), B_z = B_{nz}$ (or $B'_{nz} = B_{nz}, B_{nz} = F_z$ already)	BOTH_VALID_T_O
OPAQUE_INV	$F_z != B_{nz}$	$B_z = F_z$,queue(F) (or $B'_{nz} = F_z$)	BOTH_VALID_T_O

Table 5: State transition table for phase 3. OPAQUE_INV only occurs in phase 31, not in phase 32, phase 33, etc. Note: that for phase 3, the B_{nz} and B'_{nz} are in different physical locations depending on even or oddness of the phase. Phase2 looks like an even phase 3.

Craig Wittenbrink
NOV. 10, 1999

fragments in a pixel that are not occluded processing may complete in phase 2, and so on. Tables 3, 4, and 5 provide the state transition diagram, next state transitions as well as the side effects and outputs. In phase 3 the B_{nz} and B'_{nz} alternate their physical location as shown below in Table 6. The outputs and side effects are given in sequential order, for example in Table 3, 7th row, "VALID_TRANS, F_0, $F_z > B_z$," the frame buffer fragment is written to the fragment buffer, queue(B), its Z value saved as next z, $B_{nz} = B_z$, and then overwritten by the fragment under consideration, $B = F$.

physical location	phase 1	phase 2	phase 3 odd	phase 3 even	phase 3 odd
B_{nz}	B_{nz}	B_{nz}	B'_{nz}	B_{nz}	B'_{nz}
B_z	B_z	B'_{nz}	B_{nz}	B'_{nz}	B_{nz}

Table 6: Phase 3, physical location changing meaning from Bnz to Bnz' in odd and even.

This state machine has been fully implemented and debugged, and several models have been run through it as discussed in the next section.

4 Results

The fragment buffer implementation with a Z buffer and an extra Z buffer as described in the previous section has been implemented. Statistics have been gathered on several models to provide an indication of the performance implications. Table 7 provides statistics for processing of 4 scenes. All images were 512x512 pixels. The scenes as rendered are given in Figures 11 to 14. Artifacts from OpenGL (middle images) include the tire in the back seat of the chevy in Figure 13, and the nose gear on top of the helicopter in Figure 14.

data	no. of phase 3	total passes	Z bandwidth	frag bandwidth	avg depth
scene	6	8	2,765,189	7,497,201	2.35
spheres	10	12	8,178,293	36,889,274	3.93
chevy	8	10	3,404,954	8,396,718	2.17
heli	9	11	2,798,204	8,155,998	2.66

Table 7: Fragment buffer processing statistics, bandwidth in bytes, all 512x512 frames.

The necessary bandwidth to the memory holding the frame buffer, extra z buffer, and fragment buffer was computed during the execution of the simulation. This considers the memory model shown in Figure 2, where the frame buffer, extra Z buffer, and fragment buffer are all in the same memory. Table 7 provides the memory traffic for the conventional Z buffer (Z bandwidth), and the fragment buffer (frag bandwidth). The depth complexity is also provided in the table, and is an average over pixels covered, for the case where all geometry is considered transparent. Figure 10 plots the conventional Z buffering bandwidth to the fragment buffer bandwidth. For these scenes, it can be seen that the number of passes may be high, on the order of 8 to 12 passes. For complex interpenetrating transparent objects the depth complexity can be arbitrarily high at a given pixel. For example, in unstructured volume rendering, the depth complexity will be much higher, as there are thousands of layers for some pixel locations. But, because the application sorting of the data prior to rendering is such a burden, even the numerous passes of the fragment buffer will achieve superior results.

The key thing about the bandwidth numbers, is that they are on the same order as the Z-buffering, and there will also not be any texturing at the time that the fragments are being sorted. Therefore, the

Craig Wittenbrink
Nov. 19 1999

performance impact may be modest, because the first pass will be similar to Z-buffering, and the subsequent passes will not be competing with texture mapping operations. The ratio of Z buffering traffic to total fragment buffer traffic varies from 2.5 to 4.5 in these examples. These examples are also severe, in that all surfaces are transparent, so a new capability will place different stresses on the system. True transparency is not as easy as Z buffering, so some difference in processing is expected. The number of passes for these scenes is not expected to vary with higher resolutions, but the number of fragments will increase in both Z buffering and fragment buffer processing.

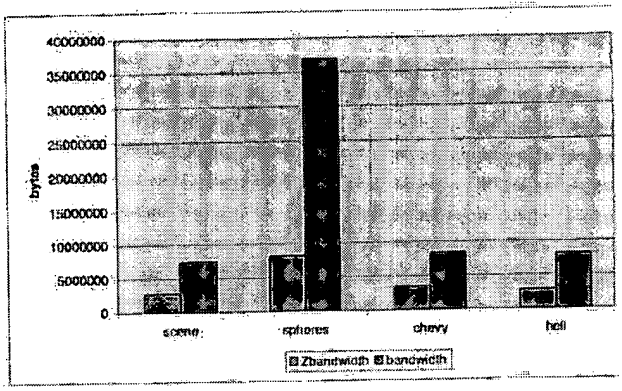


Figure 10: Bandwidth for the four scenes shown in Table 7. Conventional Zbuffering traffic in bytes is compared to rendering all surfaces as partially transparent with the fragment buffer. Those with the highest scene complexity and depth complexity require more bandwidth.

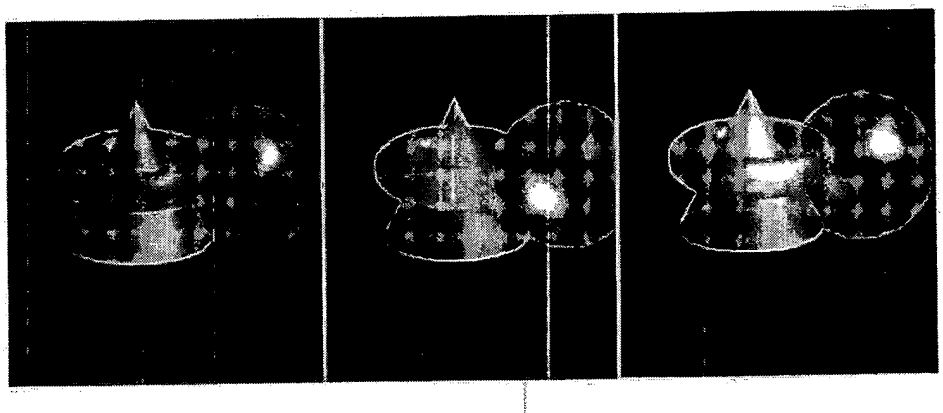


Figure 11: A scene of a cone torus, and sphere. The left image is Z buffering, the middle image is OpenGL, and the right image is the fragment buffer.

Figure 15 shows an example of the multiple passes that are taken. In the phase 1 pass, in the upper left, the rearmost fragments are determined, and placed into the frame buffer. On the next pass, in phase2, the next furthest transparent layers are composited into the frame buffer. In these renderings, the front and back faces of triangles are shaded, as the front and rear of the sphere are visible with all surfaces slightly transparent.

The implementation was also rigorously validated with permutations of cases of 3 transparent levels, as shown in the introduction, and with 3 transparent levels and an opaque layer that was placed in all possible

Craig Wittenbrink
Nov. 19 1999

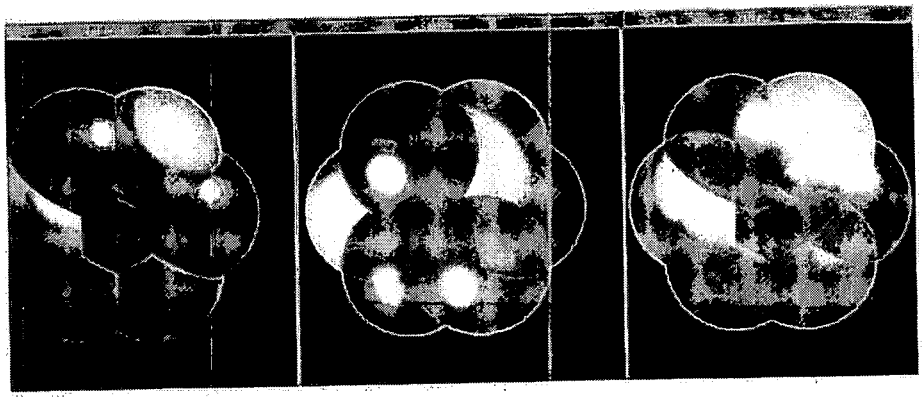


Figure 12: A scene of 7 intersecting spheres. The left image is Z buffering, the middle image is OpenGL, and the right image is the fragment buffer. This scene is meant to compare to Mammen's Figure 4, where he also renders 7 intersecting spheres.

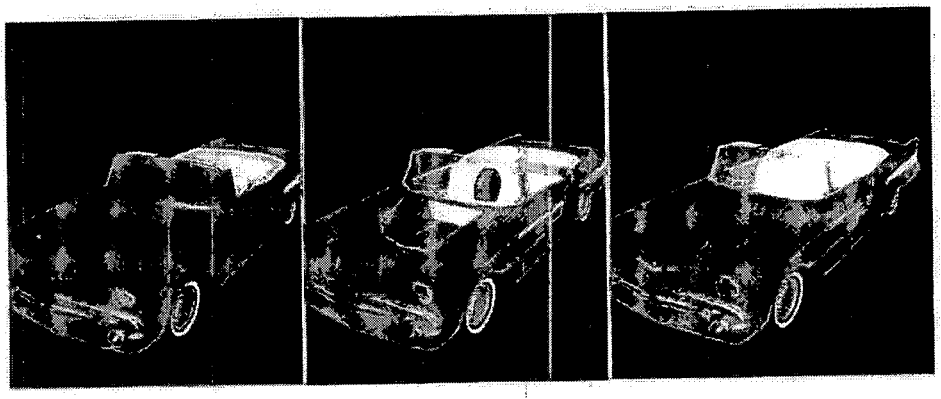


Figure 13: A scene of a Caligari True Space model of a 1957 Chevy.The left image is Z buffering, the middle image is OpenGL, and the right image is the fragment buffer.

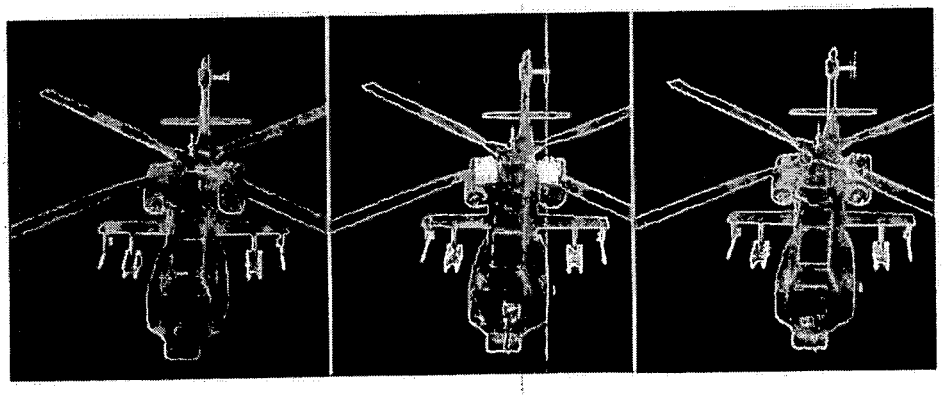


Figure 14: A scene of a Caligari True Space model of an Apache Helicopter. The left image is Z buffering, the middle image is OpenGL, and the right image is the fragment buffer.

Nov. 10, 1999

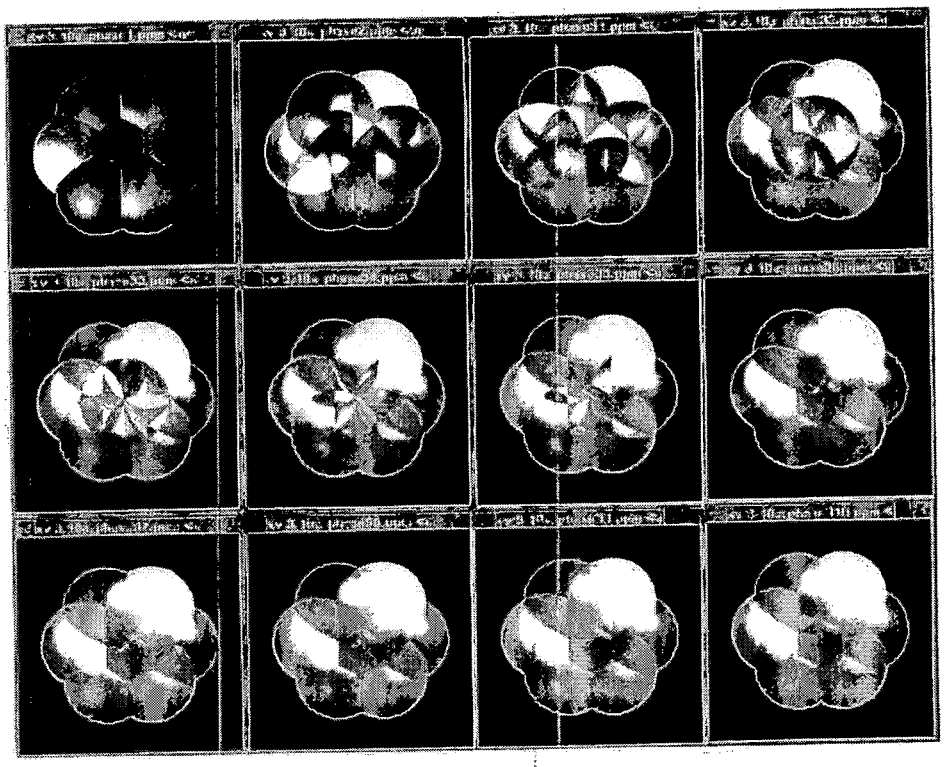


Figure 15: 12 passes taken to compute the fragment buffer with the 7 interpenetrating spheres. Passes are ordered from left to right, taken from top to bottom.

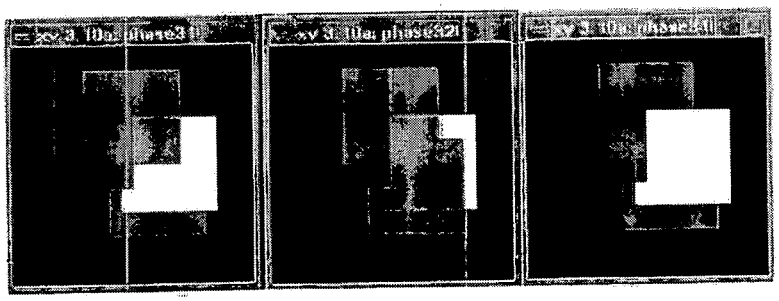


Figure 16: 3 examples of rendering test images with 3 transparent and 1 opaque layer.

Craig Wittenbrink
ABV 10, 1999

locations, in front of all three, at the same depth as all three, behind all three, or in between the 1st and 2nd or 2nd and 3rd. The transparent layers could be drawn in 6 orders. The opaque layer was placed at 7 different locations, and the opaque layer could be drawn in 4 different orders in relation to the transparent layers. Figure 16 shows 3 examples from the 168 (6*7*4) combinations that were verified.

From the simulation, conclusions can be made regarding the size of the memory needed to support this functionality. The frame buffer is assumed to contain a fragment, which I define simply as R, G, B, A, taking 5 bytes or so. Each component of RGB may be 1 byte, and alpha typically needs higher precision for compositing, which would be necessary for an architecture supporting true transparency. In the examples shown, all alpha processing is back to front, so an alpha buffer isn't actually needed in this instance but has been assumed. So, $resolution * 5$ is the frame buffer storage in bytes. Additionally, a Z buffer is used, and this is assumed to take 4 bytes. For fragment buffer processing, an additional Z buffer is used for 4 additional bytes per pixel. It is assumed that the 3 bits needed for state per pixel, can be within this 4 bytes, or simply added 4 bytes + 3 bits.

Now the actual size of the fragment buffer to be used is at least as large as the number of fragments to be stored in the fragment buffer on phase 1. This is somewhat arbitrary, as it depends on the depth complexity (DC) of the scene (fully transparent depth complexity), and the percent of the frame buffer covered. The added memory is then $DC * percent\_covered * added\_storage$. The added storage is straight forward to calculate, as it is simply the sum of a fragment's bytes (5), Z's bytes (4) , and the address of the pixel the fragment maps to (3) to total $12 = 5 + 4 + 3$. In this case 3 bytes would be able to address a 4096x4096 frame buffer. Table 7 detailed the bandwidth on external buses. Table 8 shows the amount of memory needed for these datasets. The direct measurements show that from 2.13 to 3.63 times memory is needed for a system with the fragment buffer. The resolutions can be scaled up, and the assumption that the same percentage of pixels are covered, and that the depth complexity stays the same, means the ratios are approximately the same. The only variable affected by resolution is the size of the address necessary to store on the fragment buffer itself. For 512x512 frame buffer 18 bits may be used. For slightly more bits higher resolutions are achieved, 1024x1024 with 20 bits, 1280x1024 with 21 bits, and 1600x1200 with 22 bits. This increases the ratios of additional memory needed only slightly (2.13 to 2.17 for the helicopter).

data	frame buffer	extra Z	fragment buffer	total fragment buffer	total frag/frame buffer
scene	2.25 MB	1 MB	1.57 MB	4.82 MB	2.14
spheres	2.25 MB	1 MB	4.92 MB	8.17 MB	3.63
chevy	2.25 MB	1 MB	1.85 MB	5.10 MB	2.27
heli	2.25 MB	1 MB	1.55 MB	4.80 MB	2.13

Table 8: Fragment buffer processing statistics, minimum memory usage in MBytes, for the 512x512 frame buffer (constant), extra Z (constant), fragment buffer, total of fragment buffer, frame buffer, and extra Z, and a ratio of the total fragment buffer memory versus the frame buffer for traditional z buffering.

But, for tile based hardware, there are additional improvements possible because of the reduced address to be stored with fragments on the fragment buffer. For example a 1600x1200 screen, requires 11 bits for X and Y, for a 22 bit overhead per fragment. Tiling by 64x64 tiles, requires 12 bits for X and Y, versus 22 bits, a 10 bit saving for fragments. The savings is a fixed percentage, no matter how large you decide to make the fragment buffer, ranging from 10% to 15% for the 512x512 to 2048x2048 resolutions considered with a 64x64 tile. For most graphics the depth complexity is considerably less, so less memory would be consumed.

Craig Kolb
Nov 19 1999

5 Fragment Buffer Variants

The fragment buffer implementation as described is targeted for lowest cost with reasonable performance. A continuum of choices trading off memory versus number of passes is possible. In addition the memory overhead is reduced for tile based architectures. Antialiasing can also be supported, which provides for an adaptive supersampling strategy. Table 9 shows the number of passes needed for various hardware complexities. Case 1 is 1 frame buffer, for which $2n$ passes are needed, where n is the worst case transparent layer depth complexity. Case 2 is the case just presented, where a frame buffer is used, and one extra frame of Z. The number of passes is halved to n. This case 2 also has the advantage that no explicit tile or region compositing is needed, because compositing occurs when the fragment z value matches the next z value $F_z == B_{nz}$. Case 4 is the most general, where there are N, frame buffers available for the entire screen (or tile). In this case only n/N passes are necessary, but with the obvious increase in necessary memory.

case	description	total passes
1	1 frame buffer	$O(2n)$
2	1 frame buffer and 1 Extra Z buffer	$O(n)$
3	2 frame buffers	$O(n)$
4	N frame buffers	$O(n/N)$

Table 9: Trading off memory versus number of passes. Case 2 is the one presented in the previous section.

In this section we briefly discuss the options of using fragment buffering with a tile based, deferred shading architecture we call DeferredZ, and how to support antialiasing with the fragment buffer. I show an example of processing to implement Carpenter's classical A-buffer, a software technique, in hardware. The DeferredZ architecture is a pipeline that delays the z compares and sorting, and also allows deferred shading to be performed. Figure 17 shows the overall DeferredZ architecture. From the left, primitives are sent across AGP (Accelerated Graphics Port), into the Preculling and XY bucketization stage. The output of this stage are the Screen space primitive vertex triangle strips that survive culling. The next stage is the Hierarchical Z-culling or culling, where hierarchical Z-culling is performed only for a region of the screen. Following this is XY rasterization of primitives that are not occluded. In this case rasterization means conversion of the triangle data to raster X Y coordinate location fragments, although all shading attributes such as normals are forwarded with the fragment.

The next stage is the fragment buffer compare logic as described earlier, with compositing moved later to follow shading. The Fragment Stack, is used for temporary storage to consider surviving fragments. Fragments after the first pass are sent along to the Lighting and Coloring and Shading phase of the pipeline. Because the primitives are sent in the proper order, they can be directly composited once all lighting calculations have been done. After this point primitives are sent directly into the frame buffer, which is distributed in the screen bucket fashion. It is assumed that buckets would be screen square tiles, but this architecture could also use scanline parallelism, or pixel interleaved parallelism.

There are four memories in the system, labelled A, B, C, and D. There are two new memories in B: the *Fragment Buffer* and additional Z-buffers and attribute storage. There may be none or a small number of additional Z-buffers. Figure 21 and 22 show $N = 2$, and other cases of $N = 0$, and $N = 1$ are analyzed also. The pipelines are truly independent, and no intercommunication is required between them for single pass rendering. This assumes that the texture memory is replicated so each pipeline has unfettered access to the textures that they need.

Craig Kolb Wittenbrink
NOV 10, 1999

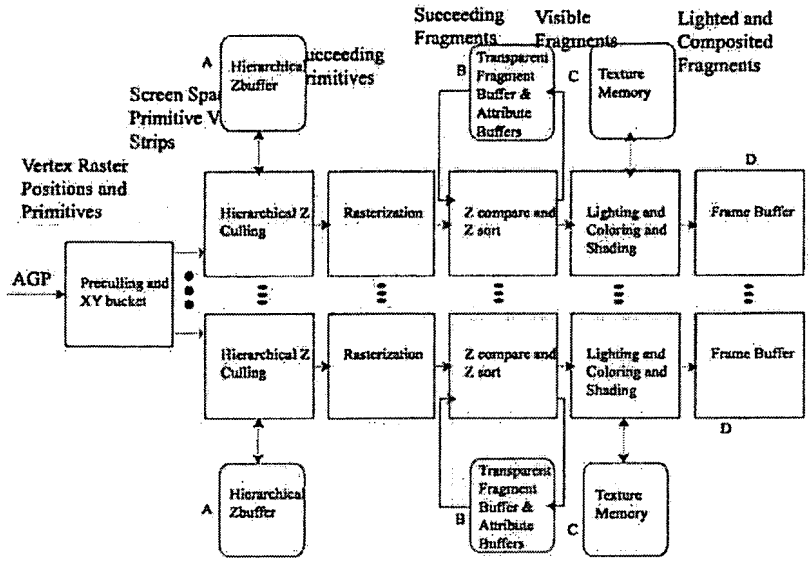


Figure 17: DeferredZ overall architecture

5.1 Fragment buffer for correct transparency

This variant of the fragment buffer provides deferred shading and true transparency. Figure 21 shows the memories including the fragment buffer, depth buffer(s), and attribute buffer(s). Additional depth and attribute memories may be used, but the rendering algorithm will be described with two Z and attribute buffers, and other cases will be described following that. The line surrounding the memories are those in Figure 17 labelled B, fragment buffer and attribute buffers. The fragment buffer may be implemented as stack, RAM, queue, or FIFO, because the order in which the fragments are considered is sequential. All fragments are read in during a phase, and may be requeued.

Referring to Figure 17, DeferredZ hardware, the XY sort portion, performs the geometric processing necessary to calculate the screen space vertex locations. The HiZ block performs hierarchical Z-buffering as explored by Ned Greene, [4]. At this point are the succeeding primitives, typically triangles defined by their vertices. The triangles enter the rasterization stage, where fragments are determined. A fragment in the DeferredZ architecture is the per sample data with Z-a numerically computed perspective depth, and the attributes, A, that are used for lighting, texturing, and shading.

Definition Fragment = Z + Attributes (including material value, normal, texture coordinates).

The processing in the blocks given in Figure 21 consists of the following:

1. Rasterization: take succeeding primitives and compute succeeding fragments.

2. Z compare/Z sort/AA compare/ AA sort: Take succeeding fragments and process them to determine the first set of visible fragments. The number of visible fragments determined depends upon the number of Z and attribute buffers, A.

3. Light/shade/texture/composite Take visible fragments, and compute the lighting, shading, and texturing, and composite with themselves and framebuffer.

The calculation performed in the Z-compare/AA compare, Z-sort/AA sort can be described as occurring in multiple phases. The first phase is where all triangles (or primitives, where primitives may include polygons, lines, points, etc.) are rasterized, and sent to the compare and sort block. The compare and sort

Craig Wittenbrink
Nov. 10, 1999

block processes them to compute the Z ordering for N layers of attributes. If N is 0, and only the frame buffer is used the algorithm is different as mentioned earlier.

Consider two scenarios to simplify the description, first the scenario where there is a single sample per pixel, and secondly the scenario where multiple samples per pixel are taken. For the first scenario, the correct ordering of transparent layers can be determined. Further consider the case where there are two Z and attribute buffers. The processing of primitives and fragments will take place in multiple phases. Figure 18 shows the pseudo-code for phase 1. Figure 19 shows the pseudo-code for phase 2. And Figure 20 shows the pseudo-code for phase 3 and higher. Figure 23 shows for a single pixel a hypothetical fragment covering and depth ordering. The eye is shown on the left, and the partially transparent fragments, T_x, are drawn as a line, while the opaque fragments are drawn as a line with cross hatching. Let us consider this as an example and describe the processing that occurs during each phase of Z fragment sorting.

To start in phase 1, Figure 18, all primitives are transformed, occlusion tested, then rasterized. As rasterized fragments are created, they are inserted into the two buffers to determine the closest opaque fragment and the furthest unoccluded transparent fragment.

```
Rasterization:
for all primitives
   rasterize to fragments
   pass fragments to Z compare and Z sort

Z compare and Z sort phase 1:
for all fragments (as received from rasterization) {
   fetch Zmin from memory
   if z>= Zmin discard fragment
   else if fragment is opaque {
      write z->Zmin,
      write fragment info into attributes for shading/lighting etc.
   }
   else /* transparent */ {
      fetch Zfar transparent, and A-attributes from memory
      if Z > Zfar transparent {
        write z-> Zfar transparent and overwrite attributes
        write Zfarold and attributes to fragment buffer <X,Y,Z, A> written
      } /* z > Zfar transparent */
      else
        write Z, attributes to fragment buffer
      } /* else transparent */
   } /* for all fragments */
```

Figure 18: DeferredZ variant, Phase 1 Rasterization and Fragment Sorting.

After phase 1 all primitives have been considered we have a processed frame buffer Z_{near}, A_{near}, $Z_{fartransparent}$, A far transparent, and a fragment buffer for all remaining fragments to be considered (X, Y, Z_f, A_f) as shown in Figure 22. Next is phase 2, to merge the remaining fragments on the fragment buffer. For phase 2, unload the current Z and A buffers to the lighting shading and compositing stage, or send Z_{near}, $Anear$ of closest opaque layer and Z_{far}, A_{far} of furthest transparent fragment. Note that in this form of the invention a valid or changed bit would be needed to know which fragments to read out, unlike the fragment buffer described earlier.

Nov 19, 1999

Figure 23 shows a case where O_1, the closest opaque layer, and $T1$, the furthest transparent layer are discarded. The remaining transparent layers are all sent to the fragment buffer. In this example, all layers beyond O_1 were either occluded or discarded, before being sent to the fragment buffer. This is not necessarily the case, as the order of primitives is arbitrary to start, so fragments that may be occluded by closer, but later arriving primitives must be occluded in pass 2. In the fragment buffer of the earlier sections this event was noted by changing state to OPAQUE_INV, which means an opaque layer invalidated previous information. For the second and all remaining phases, we will process the remaining transparent fragments using the same two Z buffers as before. Figure 24 shows the multiple phases, phase 2, phase 3, and phase 4 to sort the remaining transparent layers. So, given N Z and A or attribute buffers, it will take a number of passes $\lceil D/N \rceil$, where D is the worst case depth complexity, $D-1$ transparent layers, and 1 opaque layer. The algorithm for the phase 2 is given in Figure 19. The algorithm for the phase 3 and higher is given in Figure 20. These phases are the same in spirit as the state machine provided earlier, with the difference here being 2 full attribute and Z buffers, and a frame buffer following compositing. This aspect of the invention has not yet been implemented.

```
set Zfar transparent to -infinity
set Zmin to previous Zmin or closest transparent fragment value
for all fragments (retrieved from fragment buffer) {
   fetch Zmin from memory
   if z>= Zmin discard fragment
   else /* transparent, and not occluded */ {
      fetch Zfar transparent, and A-attributes from memory
      if Z > Zfar transparent {
         if Zfar is valid {
             if Zfirst from far is valid {
                Zfirstfrom far, A go to fragment buffer
             }
             move Zfar, Afar to Zfirstfrom far, A firstfromfar
         }
      } /* z > Zfar transparent */
      else
         write Z, attributes to fragment buffer
      } /* else transparent */
   } /* for all fragments */
}
```

Figure 19: DeferredZ variant, Phase 2, discarding further layers, and determining next $N-1$ transparent layers.

This demonstrates the logic of replacement during the fragment buffer processing. The key advantage of the fragment buffer for transparency is it computes exactly the correct ordering with fixed Z buffer, attribute buffer, and a variable fragment buffer. By only reading out fragments that have been updated, and invalidating those while writing them out you can get the valid bits for Z_{far} and $Z_{firstfromfar}$ reset very economically. The technique does not preclude image space subdivision for example, two areas being worked on, one with lighting and compositing occurring, and one with Z compare Z sort occurring to effectively load balance the work between units. Or, for example, passing along any succeeding fragments with texture values to be processed, and re-Z-buffered, depending on the mix of work. Next to be described is antialiasing

Nov 10, 1999

```
for all fragments (retrieved from fragment buffer) {
   if z > Zfar {
      if Zfar is valid {
          if Zfirst from far is valid {
             Zfirstfrom far, A go to fragment buffer
          }
          move Zfar, Afar to Zfirstfrom far, A firstfromfar
      }
      overwrite Zfar with Z
   }
   else if z > zfirst fromfar {
      if zfirst from far is valid {
         zfirstfromfar, Afirstfromfar to fragment buffer
      }
      overwrite Zfirstfromfar with Z
   }
} /* for all fragments */
```

Figure 20: DeferredZ variant, Phase 3 and higher, determining next N transparent layers.

with the same fragment buffer.

5.2 Fragment Stack for A-buffer/ Carpenter Antialiasing

An instructional example to demonstrate the flexibility of the invention is to show how the A-buffer an antialiasing scheme by Loren Carpenter [3], may be implemented. Some key things to note about the A-buffer, is that it has been often emulated, is a software only technique, and performs correct transparency as well as antialiasing. Because it is a software only technique, showing how it may be practically implemented in hardware is a significant advancement beyond the state-of-the-art. Other hardware architectures have emulated some part of Carpenter's A-buffer, such as Stephanie Winner et al. [12], but with trade-offs such as handling only a fixed number of fragments per pixel, say 4. Others such as Molnar et al. [8, 10] have claimed to be able to perform A-buffer processing, but such claims are not supported. PixelFlow cannot sort transparent layers because their compositing network, is only a Z-comparison/Z-buffering network. To do proper resorting and compositing is an unsolved problem for a PixelFlow, or sort last architecture.

A-buffer defines a fragment as a polygon clipped to a pixel boundary. They have two cases, either a pixel is fully covered by an opaque polygon, or a pixel is partially covered by transparent and/or opaque polygons. The data structure uses a linked list of fragments, sorted from front-to-back by their minimum Z, shown in Figure 26. A pixel struct stores both cases of a pixel. Figure 25 shows the pixel struct.

The mask is a 4x8 bit mask, representing subsample locations that the polygon covers. Two Z values are saved for each fragment, a minimum and maximum Z value, to aid in blending fragments that overlap. The result of processing is an array of pixel structs, the size of the frame-buffer, and linked lists of fragments of variable length for each pixel.

Now, to implement the same data structures with the fragment buffer requires sending the fragments for each pixel, on to the fragment buffer, if they are not the closest N pixels, where we have N, Z and attribute buffers. For earlier examples N is 1 or 2, and so I use, $N = 2$ for this example also. All polygons are rendered and converted to fragments, and the 2 closest to the eye fragments are stored in the random access attribute buffers, while all other fragments are sent to the fragment buffer.

Nov. 10, 1999

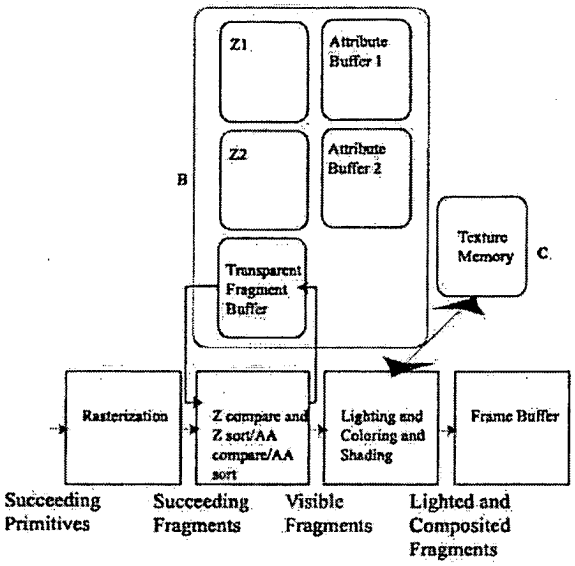


Figure 21: Memories for true transparency and improved antialiasing in the, B, fragment buffer and attribute buffers, portion.

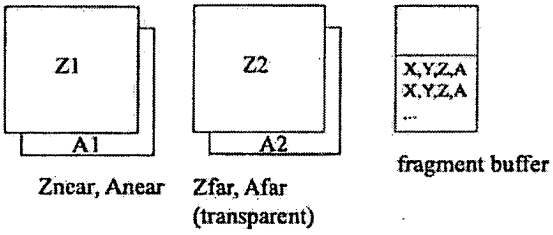


Figure 22: Buffers and fragment buffer after first phase of rasterization and sorting.

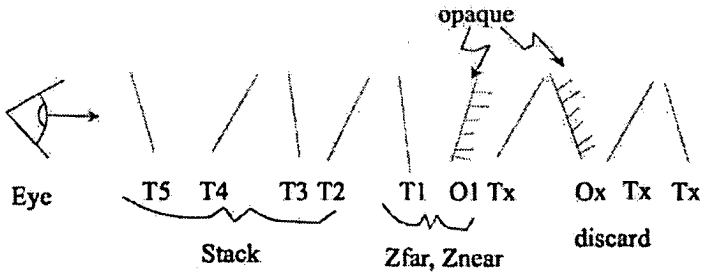


Figure 23: Example of succeeding opaque and transparent layers.

Nov 10, 1999

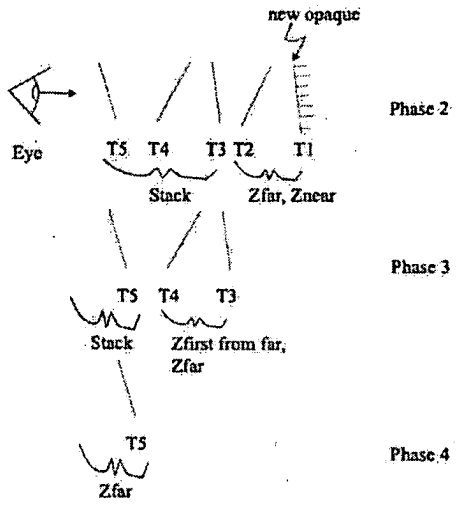


Figure 24: Remaining phases (phase 2, 3, and 4) for determining sort order for transparent layers of Figure 23

```
pixelstruct/fragment
fragment_ptr next
short_int     r,g,b
   "          opacity
   "          area
   "          object tag
pixelmask     m
float         zmax,zmin
```

Figure 25: A short int is 12 bits, and both an area and opacity are used to more accurately determine pixel coverage

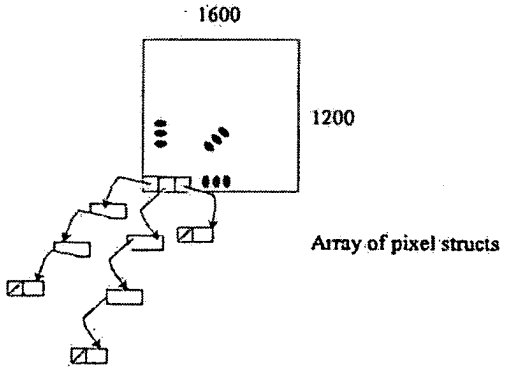


Figure 26: A-buffer data structures from Carpenter's software technique.

Nov 10, 1999

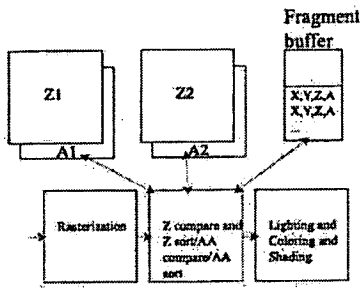


Figure 27: Fragment buffer for antialiasing, $N = 2$, Z and attribute buffers, the fragment buffer, and the hardware modules at this point in the pipeline.

Figure 27 shows a schematic of the fragment buffer, and the two Z and attribute buffers. In the case for the transparency only, we sorted from back-to-front, now, according to the A-buffer method, we sort front-to-back, by fragments minimum Z, so, in essence, we have the first two fragments for each pixel in a dedicated random access array, and all others are thrown onto the fragment buffer.

A-buffer processing requires a recursive consideration of the fragments. Instead, a multipass consideration of fragments is performed to compute the same result. Also, multiple buffer are used when traversing. First, how the front-to-back ordering is determined; the Z_1 and Z_2 buffers store the near Z's, while the attribute buffer stores the far Z, or delta Z, $Z_{min} = Z_{min}$ and $Z_{max} = Z_{min} + Z_{maxdelta}$. This allows fewer bits to be used for Z_{max}. All fragments are considered by insertion and possible replacement of Z_1 and Z_2 as described earlier. Recall the earlier example, here shown sorted after phase 1, in Figure 28.

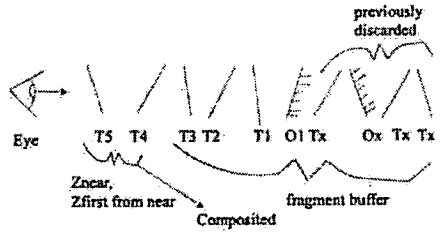


Figure 28: A-buffer example with same fragments as sorted in Figure 23.

Figure 28 shows that now we keep transparent fragments T_5 and T_4 in the Z_1 and Z_2, and all other fragments are sent to the fragment buffer. The sorting is done by nearest Z to implement A-buffer. Fragments T_3, T_2, T_1, O_1 are sent to the fragment buffer, as well as fragments beyond O_1, which would have been discarded in back-to-front ordering, T_x, O_x, T_x, T_x. We continue to sort as shown below in Figure 29.

Fragments T_3 and T_2 are captured as the next two closest fragments. All other fragments are again sent to the fragment buffer. In the next pass through the fragments, the transparent layer T_1 and the opaque layer O_1 are found, and all other fragments are discarded. The lighting and shading/compositing module blends these fragments as they, fully cover the pixel, so they are simply processed front-to-back as shown here. To show how processing proceeds when fragments partially cover a pixel, I provide example of coverage. The fragments are still all sorted front-to-back, but now, they are not necessarily composited in that order. A recursion is created by recirculating the fragments again.

A fragment's coverage of a pixel determines the inside, subsamples covered by the polygon, and outside, subsamples not covered by a polygon. Figure 30 shows the coverage of four polygons A, B, C, and D over a square pixel area. Note that improved fragment representations, such as those that use D_x, D_y slopes

Nov. 10, 1999

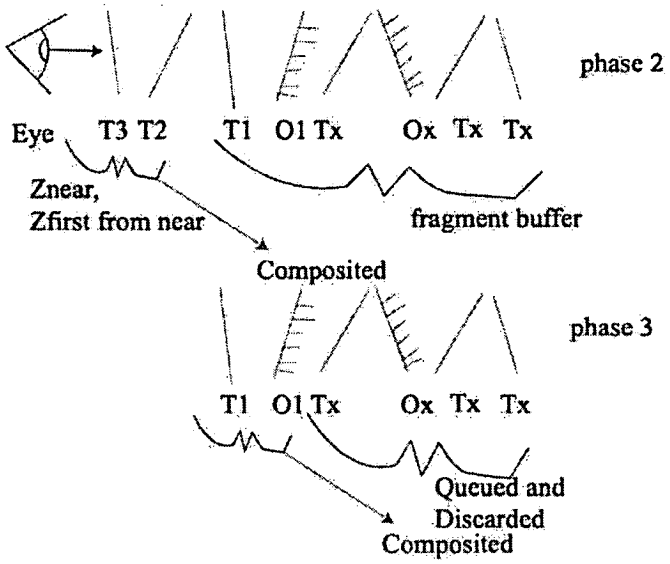


Figure 29: A-buffer example continued combining from Figure 28, continued combining the next two and the next two.

can also be used. I am using Carpenter's algorithm for clarity, and would recommend implementing a more sophisticated algorithm. The polygons would be processed into fragments for this pixel, with A, B, C, D, front-to-back after the first phase of processing. The A and B polygon fragments, would be on the Z_1, and Z_2 buffers. The masks, here shown as 4x4 would be part of the attribute buffer. When the Z_1, A_1 (attribute), and Z_2, A_2 buffers are unloaded, the search mask is computed. The search mask is the binary mask for the outside of the current accumulated fragments, M_{search}. For this example the search mask is first set to the area outside of the fragment A_{out}, $M_{search} = A_{out}$, then this search mask is used to compute in and out, from front and back of polygon fragment B. The equations directly from Carpenter, page 105.

$M_{in} = M_{search} \cap M_f$

$M_{out} = M_{search} \cap \neg M_f$

And, the interesting thing that we can do, is to change a recursive calculation to a purely iterative one by knowing that compositing can be made fully associative if computed with transparencies. The accumulated transparency is calculated in the shade/composite circuitry, and stored in the frame buffer. Essentially take the following equation from Carpenter

$C = C_{in} \times A_{in} + C_{out} \times (1 - A_{in})$ as recursed for our example $C = C_{inA} \times A_{inA} + C_{out1} \times (1 - A_{inA})$

$C_{out1} = C_{inB} \times A_{inB} + C_{out2} \times (1 - A_{inB})$

$C_{out2} = C_{inC} \times A_{inC} + C_{out3} \times (1 - A_{inC})$

$C_{out3} = C_{inD} \times A_{inD} + 0$

Here C is color, C_{in} is the color from the inside covered area of the fragment, A_{in} is the area coverage of the fragment. Th final color for the pixel is C_{out3} Here Carpenter's approach is converted to purely a sum in terms of transparency:

$C = C_{inA} \times A_{inA} + t_A \times C_{inB} \times A_{inB} + t_{AB} \times C_{inC} \times A_{inC} + t_{ABC} \times C_{inD} \times A_{inD}$

Four separate contributions from the four separate layers, where transparency of polygon fragment A is $t_A = (1 - A_{inA})$. The equations show how the recursion may be converted to direct evaluation. Figure 30 shows how the masks for coverage of each fragment are initially calculated. The A_{in} in the compositing equations are those masks, constrained by the search mask. The search mask is updated as the sorting processes fragments from front-to-back. As before fragments A, B, C, and D are sorted front-to-back, and

Nov 19 1999

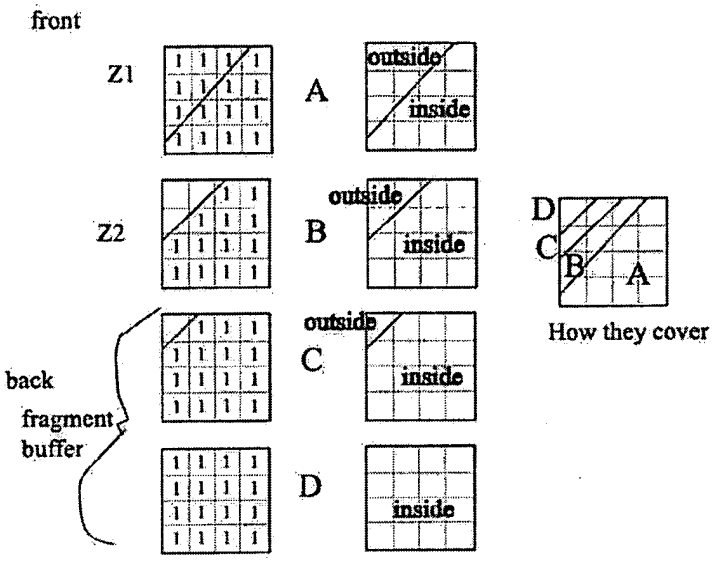


Figure 30: Four polygon's coverage masks for a single pixel.

shown in Figure 31. The search mask is updated front-to-back, and used at each step to compute the A_{in}, or as an approximation to the area. The second column $M_{in} = M_{search} \cap M_f$ is used for A_{in} at those steps. And, each A_{in} is used to update the accumulating transparency. A_{in} for $A = M_{in}$ for A, 10/16. Then $t_A = 1 - 10/16$. A_{in} for B equals $M_{inB} = 3/16$, $t_B = 1 - 3/16$, and $t_{AB} = (1 - 10/16) \times (1 - 3/16)$ $t_{ABC} = (4/16) \times (13/16) \times (14/16)$

Figure 31 shows clearly how the equation using transparencies is built up in time. I show below the pipeline processing and calculating for the final result. The current fragment and the search mask are used to compute mask in and the next search mask.

The search mask is computed by $M_S = M_{OUT} = M_S \cap \neg M_{f(current)}$ and the mask for inside the fragment is computed by $M_{incurrentfragment} = M_S \cap M_{fcurrentfragment}$. The mask calculation unit has space for 3 masks, the current fragment mask is loaded from one of the attribute buffers, A_1 or A_2. The masks are used to compute the opacity or roughly the area by summing the bits with the min mask.

When the fragment pixel location has been processed, the search mask for that location must be sorted to memory into the attribute buffer, and reloaded. The same is true for the transparency, and the accumulated color. To implement A-buffer, the search mask would have to be added to random access memory storage, as in Z_1, A_1, Z_2, A_2, and M_S for the sort and compare memories. This would allow for fragments from different pixel locations to be considered in any order, as the mask state will always be available.

6 Conclusions

The economical implementation of true transparency has been shown through the invention of the fragment buffer. A simulation was developed for the best fit for a traditional Z buffering architecture. The detailed hardware architecture, and control logic were presented. Simulation results were shown, as partial validation of the design with several models. The fragment buffer can provide true transparency with additional off chip DRAM storage, used in combination with new comparison, control, and compositing logic. Z buffering architectures already include most of the comparison logic and compositing. The novel aspects are a state machine per pixel, and multiple phases of processing recirculating fragments.

Nov 10, 1999

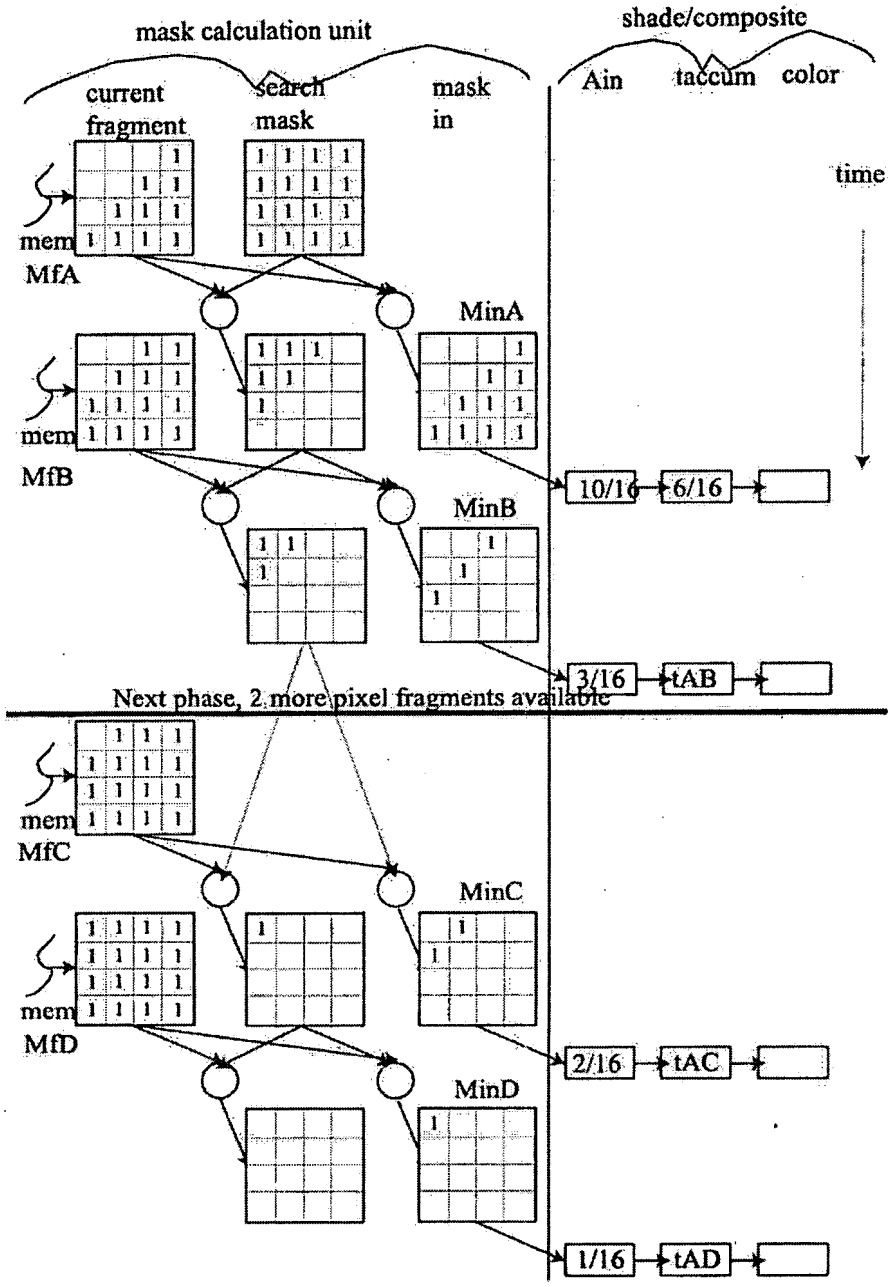


Figure 31: M_f, M_{in}, and M_{search} are computed as the fragments are sorted from front to back, shown by the timeline going from top to bottom.

Nov 10, 1999

The architecture has the advantage over previous work of 1) not requiring high per pixel dedicated storage, 2) supporting any depth complexity as long as the average depth complexity is bounded, 3) not requiring on chip storage, 4) not requiring multiple geometry passes, 5) not requiring software sorting of primitives, and 6) not being an approximation, like screen door transparency. The benefits are substantial, but much further work is needed. Examples were provided for how such a scheme could provide antialiasing, and also work in a tile based-deferred shading architecture. There is additional off chip memory required, from 2.3 to 3.6 times more memory for scenes with 12 layers per pixel and average depth complexities of 2.17 to 3.93 (for covered pixels). Because the fragment access is linear, a graceful degradation by paging fragments to system memory would provide greater capacity. The hardware logic is straightforward and can be incorporated into the current Z compare and composite of a traditional architecture. A key invention is the modification of a recursive software technique to an iterative front-to-back hardware technique. There are tradeoffs in the amount of memory used versus the required number of passes so that the ideal architecture will depend on the price point and available semiconductor technology.

References

[1] K. Akeley. RealityEngine graphics. In *Proceedings of SIGGRAPH*, pages 109–116, Anaheim, CA, Aug. 1993. ACM.

[2] S. J. Baker, D. A. Cowdrey, G. J. Olive, and K. J. Wood. Image generator for generating perspective views from data defining a model having opaque and translucent features. United States Patent Number 5,363,475, Nov. 8 1994.

[3] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, pages 103–108. ACM, July 1984. Vol. 18, No. 3.

[4] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proceedings SIGGRAPH*, pages 231–238, Anaheim, CA, Aug. 1993. ACM SIGGRAPH.

[5] M. Kelley, K. Gould, B. Pease, S. Winner, and A. Yen. Hardware accelerated rendering of CSG and transparency. In *Proceedings of SIGGRAPH*, pages 177–184, Orlando, FL, July 1994. ACM.

[6] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, July 1989.

[7] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[8] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. In *Proceedings of SIGGRAPH '92*, pages 231–240, Chicago, IL, July 1992. ACM.

[9] T. Porter and T. Duff. Compositing digital images. In *Computer Graphics*, pages 253–259, Aug. 1984.

[10] J. Poulton, S. Molnar, and J. Eyles. Architecture and apparatus for image generation. United States Patent Number 5,388,206, Feb 1995.

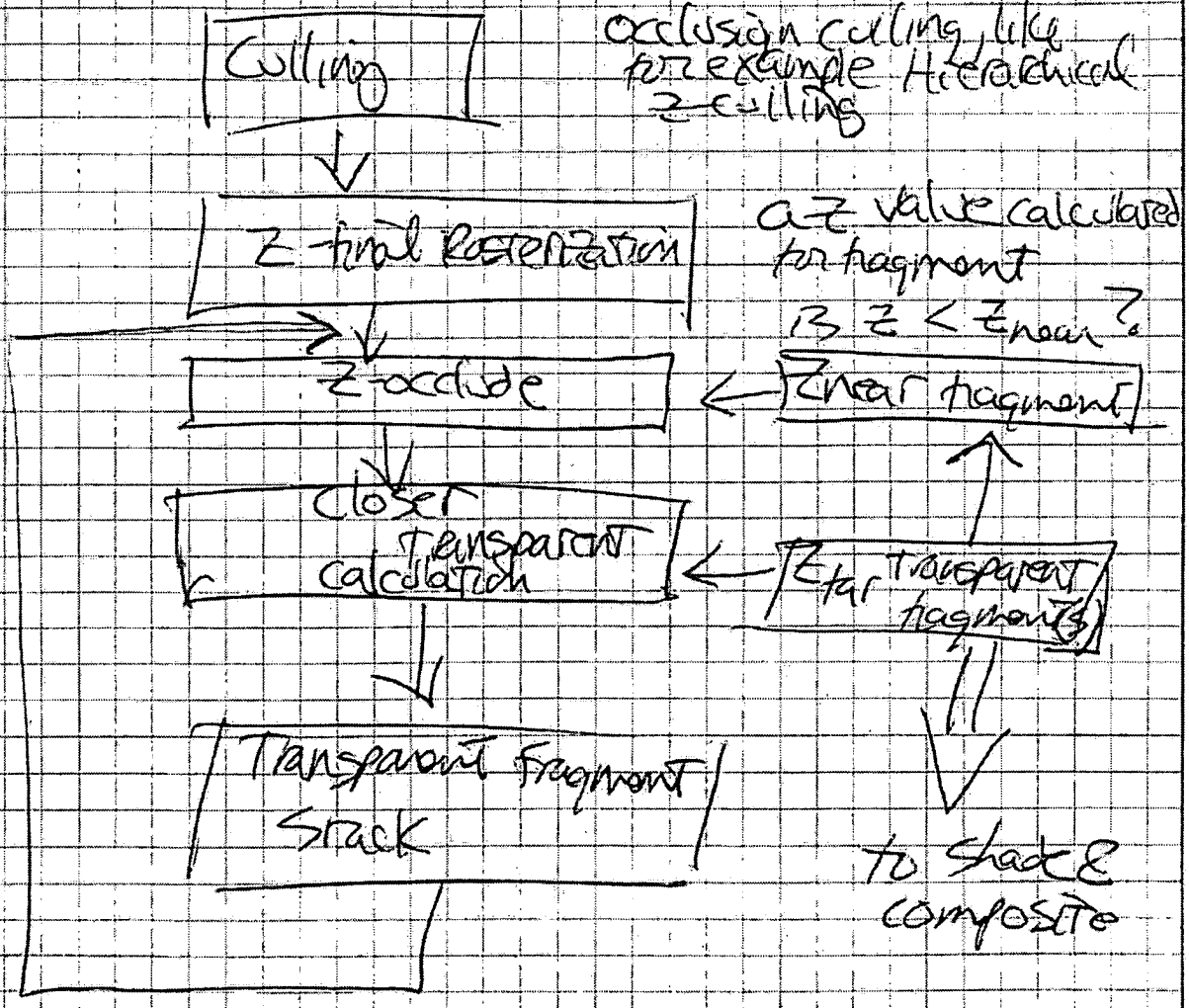
[11] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, Mar 1974.

[12] S. Winner, M. Kelley, B. Pease, and A. Yen. Hardware accelerated rendering of antialiasing using a modified a-buffer algorithm. In *Proceedings of SIGGRAPH*, pages 307–316, Los Angeles, CA, Aug. 1997. ACM.

Aug. 31, 1998

Proper Transparency calculation in hardware

A method for calculating proper transparency in hardware to make multiple passes upon the unoccluded transparent fragments. The basic approach is executed in the following pipeline:



The Transparent fragment stack stores all fragments that are not occluded $z_c < z_{near}$, and updates the K

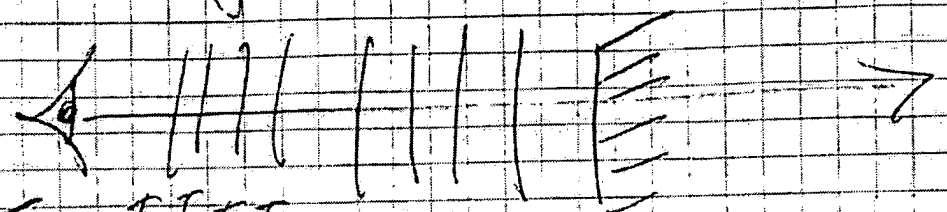
Continued on Page 18

Read and Understood By

Signed	Date	Signed	Date
Thomas Malzbender	12/8/98	Mathew E. Brown	12/15/98

Aug 31, 1998 cont.

closest to the opaque layer. For example see the Figure 18-1 below:



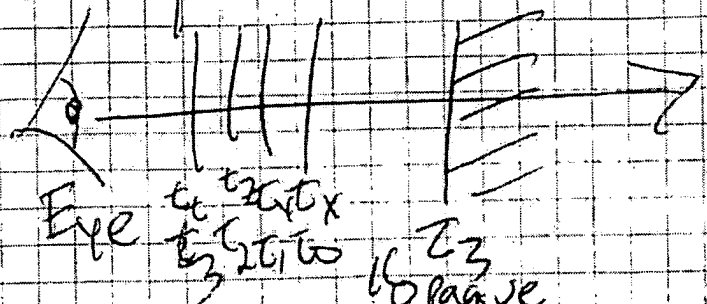
Eye $t_6 t_5 t_4 t_1 t_x$ $t_3 t_2 t_7$ t_0 opaque

store on stack — sorted

Fig 18-1

Transparent layers t_3, t_2, t_4, t_6 can be sorted on the fly in an insertion sort as all primitives are considered. Remaining layers are forwarded to the transparent fragment stack: $t_6 t_5 t_4 t_1 t_x$.

On the next pass the "occluding" layer is updated to the K+1th closest transparent layer. In this example t_3 is the remaining layer. All layers will be sorted on the next pass. Because in each pass, K layers will be sorted, all layers will be sorted in $\lceil d/K \rceil$ passes.



Eye $t_6 t_5 t_4 t_1 t_x$ $t_3 t_2 t_7 t_0$ t_3 opaque

Fig 18-2

The d is the depth complexity in terms of transparent polygons of the scene.

Continued on Page

Read and Understood By

Thomas Malzbender — 12/8/98 — Signed — Date

Michael C. Kopp — 12/15/98 — Signed — Date

Aug 31, 1998 cont.

pseudo code for the processing is as follows:

```
for all buckets
    for all primitives in bucket
        fragmentize (see OpenGL manual spec)
        Do occlusion test

for all buckets
    while transparent fragments left
        [ for all transparent fragments in stack
            - occlude
            - check for K closest to opaque
        [ send K fragments to shade and composite
          reset "opaque" layer to Kth closest layer
```

O(P/B)

O(P/B * ⌈d/k⌉)

Craig Wittenbrink

Aug 31, 1998

Sep 1, 1998 The hardware for proper transparency calculation may proceed in back-to-front operas described or in front-to-back order, so that occlusion termination may be calculated. In the previous example, Fig 18-1 and Fig 18-2 the processing would reverse, and the 4 layers closest (here K==4) to the eye would be saved in sorted order and the remaining 4 layers would be placed on the queue. In this way the accumulated opacity of the pixel can be used to terminate the calculation.

When $\alpha_{pixel} \geq$ threshold, terminate processing

Craig Wittenbrink Sep 1, 1998

Continued on Page

Read and Understood By

Signed	Date	Signed	Date
12/8/98		12/15/98	

The complexity of implementing floor() is small but ceil() requires more work, basically a floor() plus additional to see if the value was an INT to begin with.

3 main approaches to zero-pixel triangle culling were explained. The most efficient N-code approach appears to be the ceil(), floor() calc. and an efficient hardware circuit was given.

Craig A Wittenbrink

Craig M. Wittenbrink, Sep 9, 1998

December 3, 1998 Craig M. Wittenbrink

Invention for Antialiasing and proper Transparency calculation in hardware. This entry is an improvement and further details of the idea explored in the Aug 31 entry on page 2373-17 TO 2373-19.

Overview: The invention relates to the proper calculation of the ordering of transparent polygons in a computer graphics rendering and also allows for an economical means for antialiasing in computer graphics rendering.

Assumptions: Triangles are drawn in any order, and the system determines what the proper visible pixels are. The overall architecture could be one like my proposed deferred Z architecture presented internally. Deferred Z architecture is a pipeline that proposes to delay the Z compares and shading and also allows deferred shading to be performed.

The next page shows an overview of the Deferred architecture

Witnessed and understood by:

Thomas Malzbender 12/8/98 Date

Michael E. Rose 12/15/98 Date

Witness Signature Date

AA-proper Transparency cont.

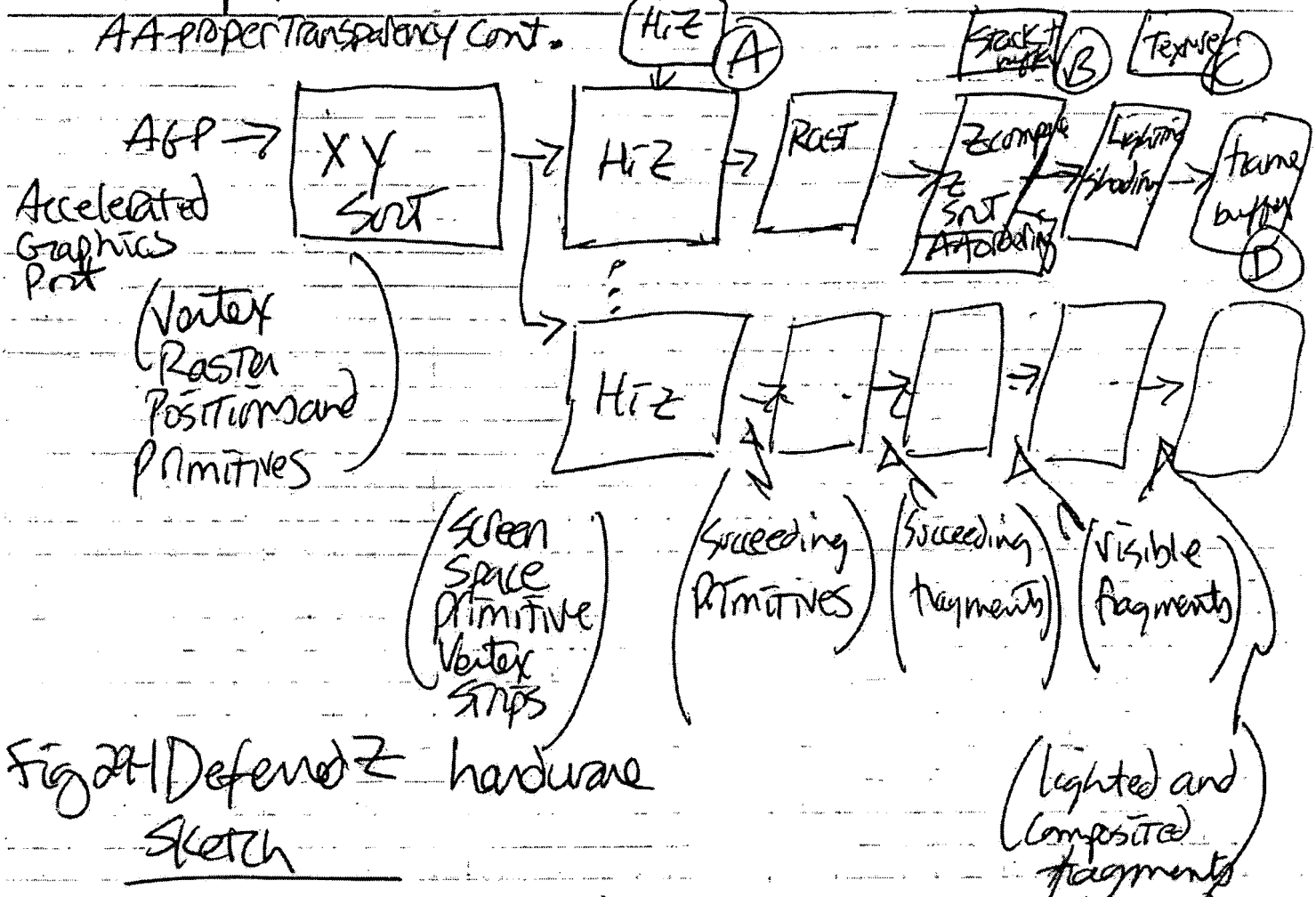
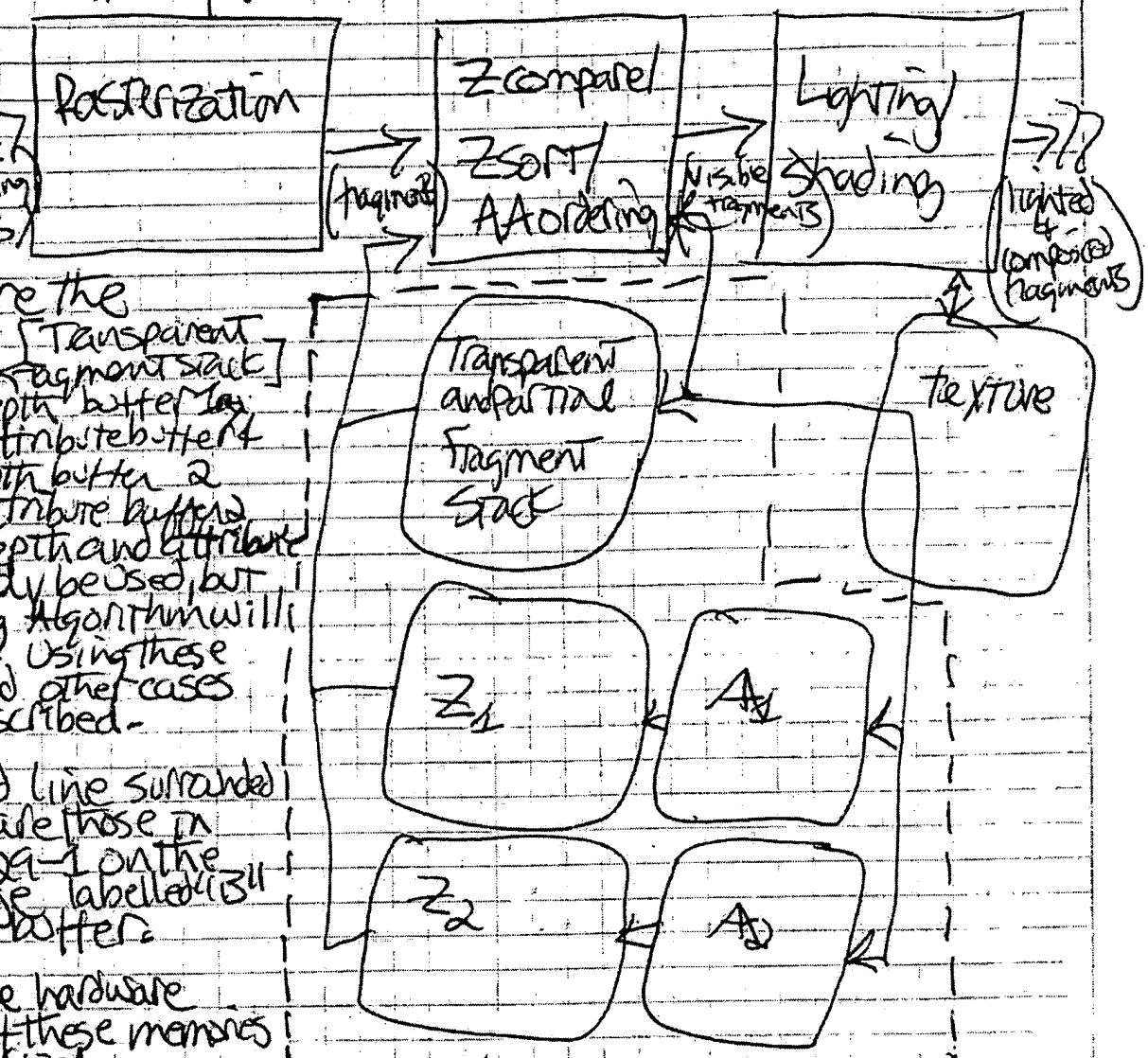


Fig 24 | Defered Z hardware sketch

There are 4 memories in the system labelled (A) (B) (C) & (D). The memories for the invention that are new are the (B) memories that include a new [Transparent or Partial Fragment Stack] and additional Z-buffers and attribute storage (from 0 to N, where N is any number, typically small). The 3 blocks of Rasterization, (Rast), Z compare/Z sort, and lighting/shading will now be investigated 30

Thomas Malzbender 12/8/98 Michael E. Goss 12/15/98

A-A proper Transparency cont.



Shown are the memories [Transparent and partial fragment stack]
Z_1 - the depth buffer
A_1 - the attribute buffer
Z_2 - the depth buffer 2
A_2 - the attribute buffer 2
Additional depth and attribute memories may be used, but the rendering Algorithm will be described using these memories and other cases will be described.

The dashed line surrounded memories are those in the figure 29-1 on the previous page labelled "B" stack and buffer.

Next the hardware functioning of these memories will be specified.

Figure 30-1 memories for True transparency and improved antialiasing, in the "B" = stack and buffer, portion of the pipeline shown in Figure 29-1.

Continued on Page 31

Read and understood By

Roman Maliborski 12/8/98 Signed Date

Michael E. Bynon 12/14/98 Signed Date

AA proper Transparency cont.,
The hardware memory buffers are used repeatedly in slightly different fashion to process all fragments generated from the rasterization phase, until the proper visible, and/or correctly sorted Transparent fragments have been determined.

Referring to figure 29-1, Deferred z hardware, The XY sort portion performs the geometry processing necessary to calculate the screenspace vertex locations The HiZ block performs hierarchical Z-buffering as explained by Ned Greene, SIGGRAPH 93, with M. Kass and G. Miller, Hierarchical Z-buffer visibility Pages 231-238, Aug. 1993. At this point are the succeeding primitives, typically Triangles defined by their vertices.

The Triangles enter the rasterization stage, where fragments are determined. A fragment, in this invention is the per sample data with Z - a numerical computed perspective depth, and the attributes A, that are used for lighting, texturing, and any other type of processing that can be used to calculate the visible pixel color.

Fragment $\triangleq$ Z + Attributes (including material value, normal, Texture coordinates))

The processing in the blocks given in Figure 30-1 consists of the following

Rasterization: Take succeeding primitives and compute succeeding fragments

Z compare/ AA compare & Z sort / AA sort: Take succeeding fragments and process them to determine the first set of visible fragments. The number of visible fragments determined depends upon the number of Z and Attribute buffers A.

Light / Shade / Texture Composite: Take visible fragments, and compute the lighting, shading and texturing, and composite with themselves and framebuffer

Continued on Page 32

Read and Understood By

Thomas Malyska 12/8/98 Signed Date

Michael E. Sim 12/15/98 Signed Date

AA proper Transparency cont.

The calculation performed in the Z compare/AA compare, Z sort/AA sort can be described as occuring in multiple phases. The first phase is where all triangles (or primitives, where primitives may include polygons, lines, points, etc.) are rasterized and sent to the compare and sort block. The compare and sort block processes them to compute the Z ordering for N layers of attributes. If N is 0, and only the frame buffer is used the algorithm is changed somewhat, but I will return to that later.

Consider two scenarios to simplify the description, first the scenario where there is a single sample per pixel, and secondly the scenario where multiple samples may be taken per pixel.

For the first scenario the correct ordering of transparent layers can be determined.

Algorithms for Rasterization and Z compare and Z sort

```
Rasterization   { for all primitives
                      Rasterize to fragments
                      pass fragments to Z compare
                        and Z sort

Z compare and
Z sort phase 1  { for all fragments (as received from
                    Rasterization) {
                    fetch Z_min from memory
                    if Z >= Z_min (of pixel) discard fragment
                    else if fragment is opaque {
                        write Z => Z_min
                        write fragment into
attributes for shading / lighting / coloring etc) overwrites previous
                    }
                    else { /* transparent */
```


Thomas Malzbender 12/8/98 Michael C. [illegible] 12/15/98

A-A proper Transparency cont.

fetch Z_{far} & Transparent, and A-Attributes from memory

if $Z > Z_{far\ Transparent}$ {

write $Z \Rightarrow Z_{far\ Transparent}$, and overwrite attributes

write Z_{far} and A-Attributes to fragment stack $\langle X, Y, Z, A \rangle$ written.

} /* ~~else transparent~~ */

if $Z < Z_{far\ Transparent}$

else write Z_1 Attributes to fragment stack

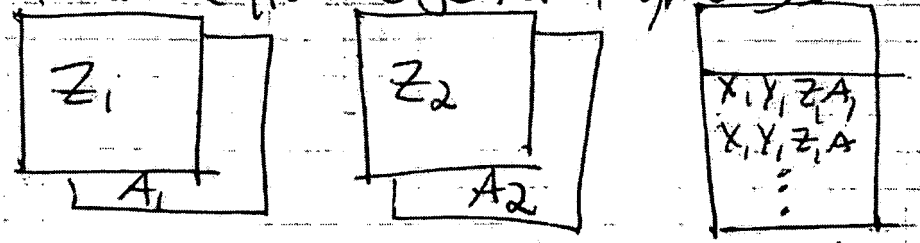
} /* else Transparent */

} /* for all fragments */

Z compare and Z sort phase 1

After all primitives have been considered we have a processed framebuffer Z_{near}, A_{near}, $Z_{far\ Transparent}$, ~~A~~ $A_{far\ Transparent}$, And a stack of all remaining fragments to be considered $\langle X_f, Y_f, Z_f, A_f \rangle$, as shown below.

Next is phase 2, to merge the fragments ~~with~~



Z_{near}, A_{near} $Z_{far}, A_{far\ Transparent}$

Figure 33-1

Continued on Page 34

AA proper transparency cont.

For phase 2, unload the current Z and A buffers to the lighting/shading and compositing stage. Take the Z_{far} Transparent (or now be the opaque Z to occlude further fragments

Send Z_{near} A_{near} of closest opaque are sent

and Z_{far}, A_{far} of furthest transparent are sent

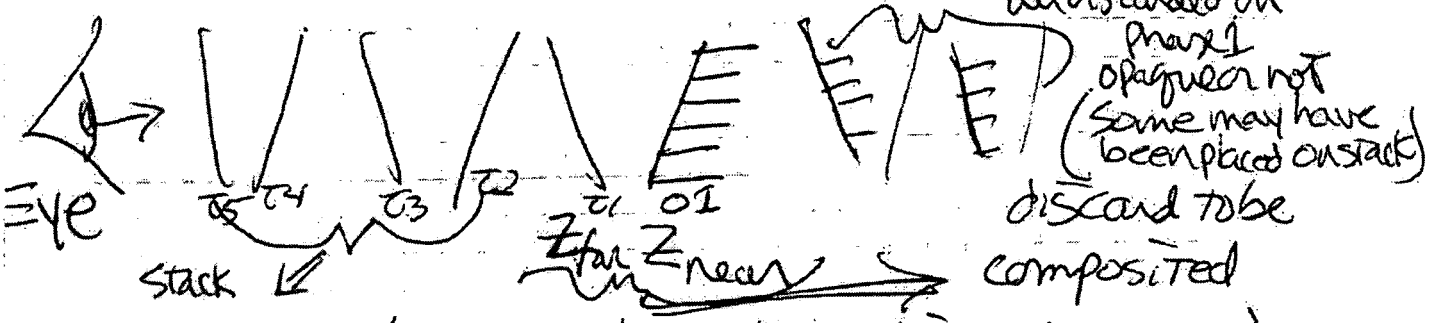


Fig 34-1 Example of succeeding opaque and transparent layers

Figure 34-1 shows a case where O1, the closest opaque layer, and O4 the furthest Transparent layer are discarded. The remaining Transparent layers are all sent to the fragment stack, so for this pixel location X_p, Y_p we have (in no particular order) The fragments in the stack.

X_p, Y_p, Z_2, A_2
X_p, Y_p, Z_4, A_4 ← fragment from different location
X_0, Y_0, X, X
Y_p, Y_p, Z_3, A_3
X_p, Y_p, Z_5, A_5

35

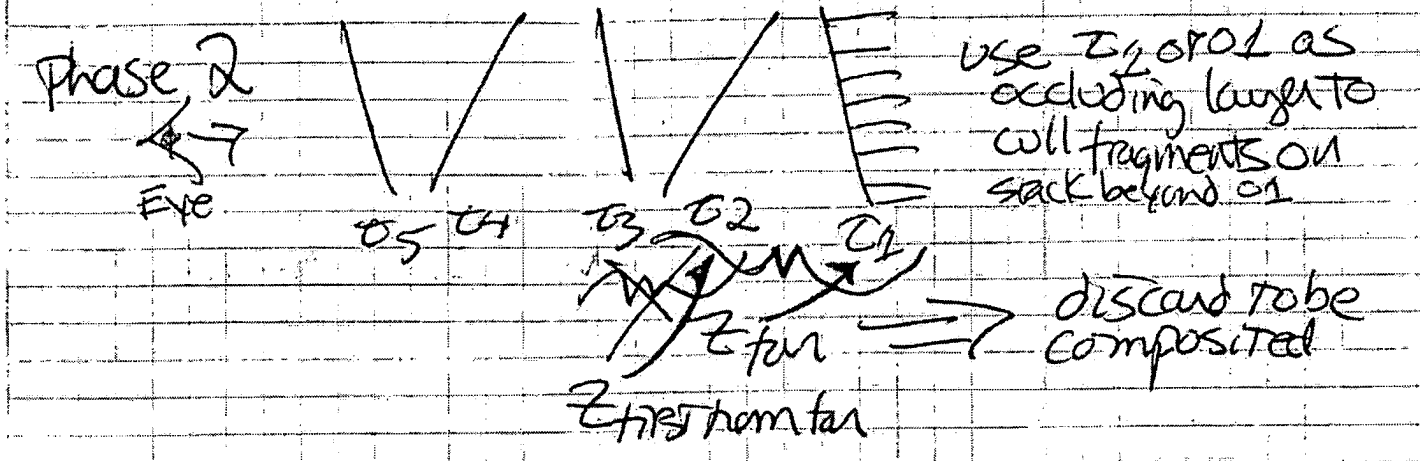
Thomas Malzbender 12/8/98 Michael C. Gross 12/15/98

PROJECT

Continued From Page 34

A Proper Transparency Control

For the second and all remaining phases, we will process the remaining TRANSPARENT fragments, using the same TWO Z buffers as before



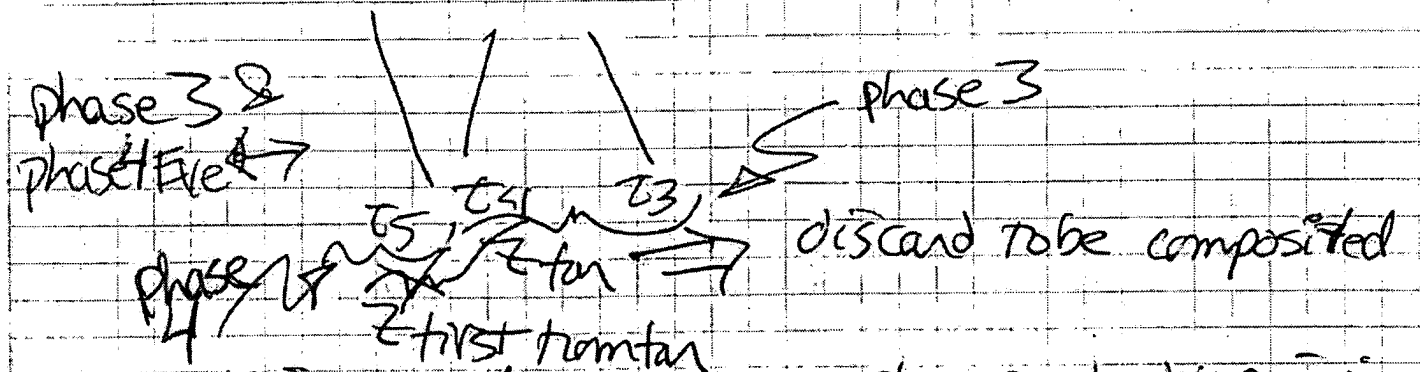


Figure 35-1 remaining phases for determining sort order for transparent layers of Figure 34-1.

So, given N Z and A buffers (attributes?) it will take a number of passes D/N passes, where D is the worst case depth complexity D-1 transparent layers and 1 opaque layer. The Algorithm for the phase 3 and higher is given on the next page (phase 2 processing is similar to phase 1)

Continued on Page 36

Read and Understood By

Thomas Malzbender	12/8/98	Michael E. Goss	12/15/98
Signed	Date	Signed	Date

AA proper transparency cont:

```
if Z > Z_far {
    if Z_far is valid {
        if Z_first_from_far is valid {
            Z_first_from_far, A go to fragment stack
        }
        move Z_far, A_far to Z_first_from_far A_first_from_far
    }
    overwrite Z_far with Z
}
else if Z > Z_first_from_far {
    if Z_first_from_far is valid {
        Z_first_from_far, A_first_from_far to stack
    }
    overwrite Z_first_from_far with Z
}
```

This demonstrates logic of replacement during the fragment stack processing. Code is not shown for Phase 2 - See Figure 35-1.

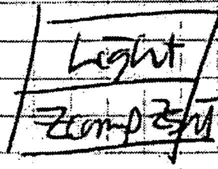
The key advantage of the invention for transparency is it computes exactly the correct ordering with a fixed Z buffer, Attribute buffer storage, and a variable fragment stack. By only reading out fragments that have been updated, and invalidating those while (Cont'd)

Continued on Page

Thomas Malzbender 12/8/98 Michael C. Gorr 12/15/98

Signed Date Signed Date

At Proper Transparency cont'd
writing them out you can get the valid bits for Zfar and Z first from fan resets very economically.
The invention does not preclude imagespace subdivision



for example two areas being worked on, one with lighting and compositing occuring (Top area) and one with Z compare Zsort occuring to effectively load balance the work between units, or, for example, passing along any succeeding fragments with texture values to be processed, and re-Z-buffered, depending on the mix of work.
Next to be described is antialiasing with the same fragment buffer, then generalization of the invention.

Craig M. Wittenbrink

Craig M. Wittenbrink

Dec. 4, 1998

December 9, 1998 FRAGMENT ~~BUFFER~~ STACK FOR ANTI-ALIASING
This invention shows how antialiasing may be performed using the Fragment stack described in the Dec. 3, 1998 entry, pages 2373-28 to 2373-37.

Continued on Page 38

Read and Understood By

Thomas Malzbender	12/8/98	Michael C Goss	12/15/98
Signed	Date	Signed	Date

Dec 9, 1998 cont

CONT. FRAGMENT STACK FOR ANTIALIASING

An instructional example to demonstrate the flexibility of the invention is to show how the A-buffer, an antialiasing scheme by Loren Carpenter, may be implemented.

[Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface method", in Proceedings of SIGGRAPH 1984, July, pages 103-108.]

Some key things to note about the A-buffer, is that it has been often emulated, is a software only technique, and performs correct transparency as well as antialiasing. Because it is a software only technique, showing how it may be practically implemented in hardware is a significant advancement beyond the state of the art. Other hardware architectures have emulated some part of Carpenter's A-buffer, such as Stephanie Winner et al, [WINN97], but with trade-offs such as handling only a fixed number of fragments per pixel, say 4. Others such as

[WINN97] Stephanie Winner, Mike Kelley, Brent Pease, Bill Rivard, and Alex Yen, "Hardware Accelerated Rendering of Antialiasing Using A modified 'A-buffer Algorithm'", in proceedings of SIGGRAPH 97, Los Angeles, CA Aug. 1997, pages 307-316.

Molnar et al. [MOLN92][MOLN95] have claimed to be able to perform A-buffer processing, but such claims are not supported. PixelFlow cannot sort transparent layers because the "compositing" network is only a Z-comparison/Z-buffering network. To do proper resorting and compositing is an unsolved problem for a PixelFlow, sort last architecture.

[MOLN92] Steven Molnar, John Eyles, and John Poulton, "PixelFlow: High speed Rendering Using Image Composition" In Proceedings of SIGGRAPH 1992 Chicago, IL, pages 231-240, July, 1992.
[MOLN95] John Poulton, Steven Molnar, and John Eyles "Architecture and Apparatus for Image Generation", US Patent # 5,388,206, Feb 1995

Continued on Page 39

Read and Understood By

Thomas Malzbender — 12/14/98 — Signed — Date

Michael C. Stone — 12/15/98 — Signed — Date

CONT. FRAGMENT STACK FOR ANTIALIASING Dec 9, 1998 cont.

A-buffer defines fragment as a polygon clipped to a pixel boundary. They have two cases, either a pixel is fully covered by an opaque polygon, or a pixel is partially covered by transparent and/or opaque polygons. The data structure uses a linked list of fragments, sorted from front-to-back by their minimum Z. A pixel struct stores both cases of a pixel

```
                    fragment-ptr  next
PixelStruct/        short int     r,g,b          A short int is 12 bits,
fragment            "             opacity        and both an area and opacity
                    "             area           are used to more accurately
                    "             object tag     determine the pixel coverage
                    pixelmask     m
                    float         zmax, zmin
```

The mask is a 4X8 bitmask, representing subsample locations that the polygon covers. Two Z values are saved for each fragment, a min and max Z value, to aid in blending fragments that overlap.

The result of processing is an array of pixel structs, the size of the frame-buffer, and linked lists of fragments of variable length for each pixel:

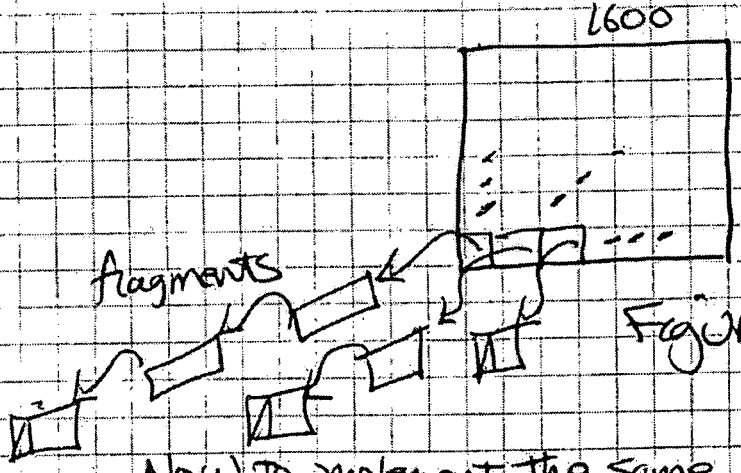


Figure 39-1 A-buffer Data Structures

Now to implement the same data structures with the fragment stack requires sending the fragments for each pixel onto the stack, if they are not the closest N pixels, where we have N, Z and attribute buffers. For the earlier examples N is 2, and so I use N=2 for this example also. All polygons are rendered and converted to fragments, and the 2 closest to the eye fragments are stored in the random access frame buffer, while all other fragments are sent to the stack. Cont.

Continued on Page 40

Read and Understood By

Thomas Malzbender 12/14/98 Signed Date

Michael C. Buzzi 12/15/98 Signed Date

CONT. FRAGMENT STACK FOR ANTIALIASING Dec. 9, 1998 cont'd

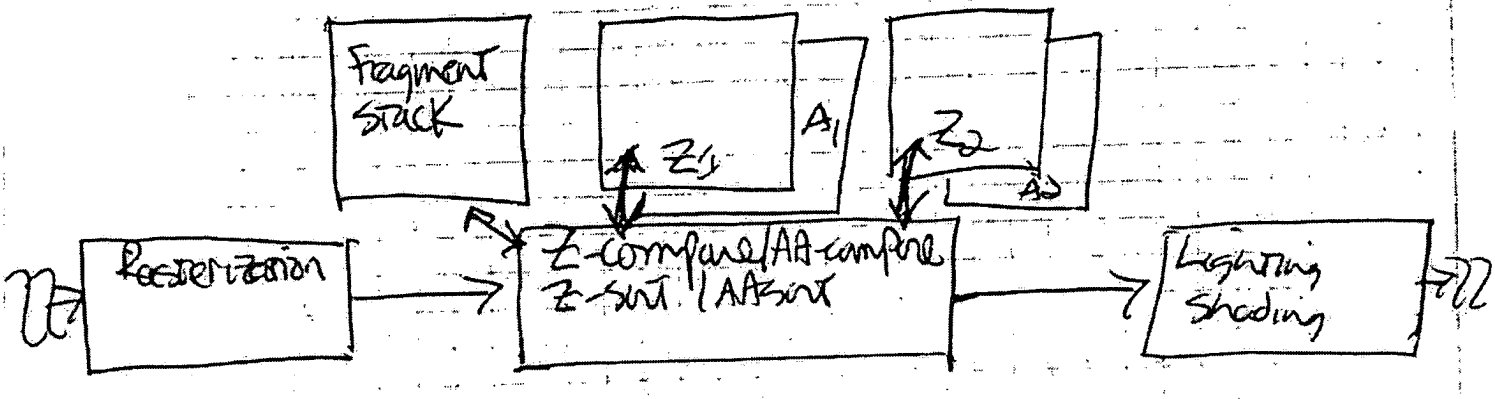


Figure 40-1 Fragment stack for antialiasing, as in Figure 30-1, 33-1. The N=2 Z and attribute buffers, the fragment stack, and the hardware modules at this position in the pipeline.

Figure 40-1 shows a schematic of the fragment stack, and the two Z and attribute buffers. In the case for transparency only, we sorted from back-to-front, now, according to the A-buffer method, we sort front-to-back, by fragment minimum Z. So, in essence, we have the first two fragments for each pixel in a addicted random access array, and all others are thrown onto the stack.

A-buffer processing requires a recursive consideration of the fragments. Instead, a multipass consideration of fragments is performed to compute the same result. Also, multiple stacks are used when traversing. First, how the front-to-back ordering is determined; the z_1 and z_2 buffers store the near Z's, while the attribute buffer stored the far z_1 or a delta Z $z_{min} = z_{min}$ and $z_{max} = z_{min} + z_{maxDelta}$. This allows fewer bits to be used for z_{max}.

All fragments are considered by insertion and possible replacement of z_1 and z_2 as described earlier. Recall the earlier example Fig 34-1

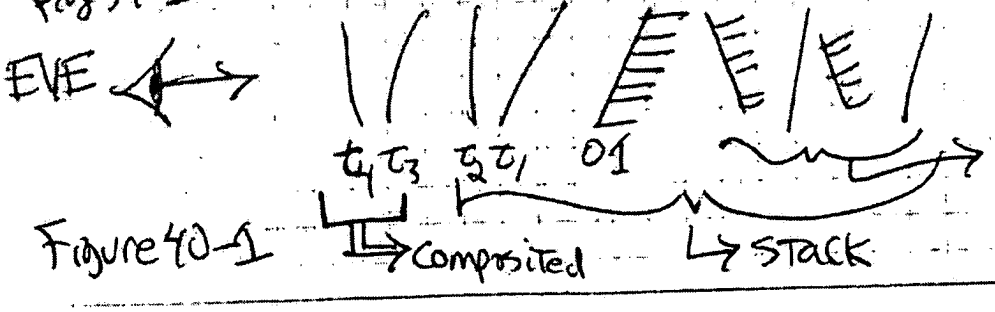


Figure 40-1

41

Thomas Malzbender 12/14/98 Michael C. Gorman 12/15/98

CONT. FRAGMENT STACK FOR ANTI-ALIASING Dec. 9, 1998 cont.

Figure 40-1 shows that now we keep Transparent fragments c_4 and c_3 in z_1 and z_2 and all other fragments are sent to the stack. The sorting is done by nearmost z to implement A-buffer. Fragments c_2, c_1, o_1 are sent to the stack, as well as fragments beyond o_1, which would have been discarded in back-to-front ordering. We continue to sort as shown below in Figure 41-1. Fragments c_2 and c_1 are captured as the next two closest fragments. All other fragments are again

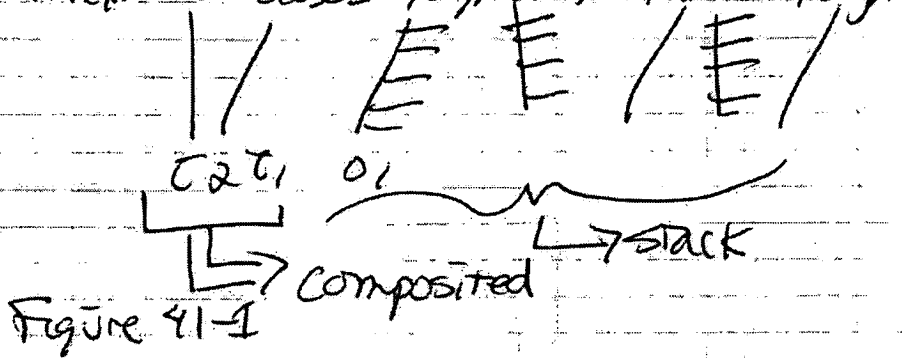


Figure 41-1

sent to the stack. In the next pass through the fragments, the opaque layer o_1 is found, and all other fragments are discarded

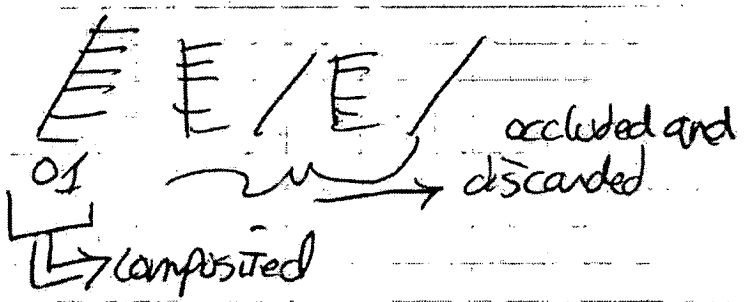


Figure 41-1

The lighting and shading/compositing module blends these fragments as they fully cover the pixel, so they are simply processed front-to-back as shown here.

To show how processing proceeds when fragments partially cover a pixel, I provide example of coverage. The fragments are still all sorted front-to-back, but now the are not necessarily composited in that order. Fragments A recursion B created by recirculating the primitives again.

42

Thomas Malzbender 12/10/98 Michael C Sun 12/15/98

CONT, FRAGMENT STACK FOR ANTIALIASING Dec 9, 1998

A fragment's coverage of a pixel determines the inside, subsamples covered by the polygon, and outside, subsamples not covered by a polygon. Shown below is the coverage of four polygons A, B, C, & D over a square pixel area.

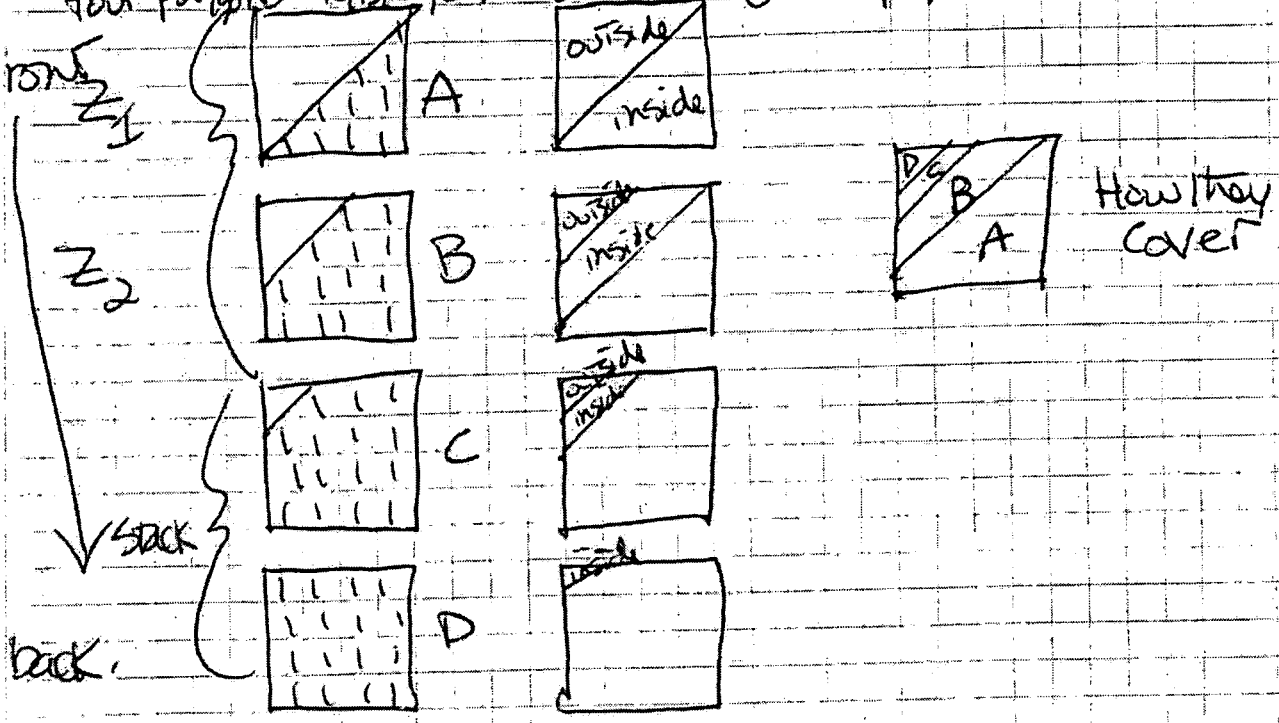


Figure 42-1 four polygons coverage masks for a single pixel

The polygons would be processed into fragments for this pixel with A, B, C, D, front to back, after the first phase of processing. The A & B would be on the z_1 and z_2 buffer, the masks, here shown as 4x4 would be part of the attribute buffer.

When the z_1, A_1, and z_2, A_2 buffers are unloaded, the search mask is computed. The search mask is the binary mask in the out side of the current accumulated fragments, Msearch.

For this example the search mask is first set to A_{OUT}. Msearch = A_{OUT}, then this search mask is used to compute in and out from front and back of polygon fragment B. The equations directly from Carpenter, page 105.

Continued on Page 43

Witnessed By

Thomas Malzbender 12/14/98
Signed Date

Michael C. Lyon 12/15/98
Signed Date

GRAPHICS

CONT. FRAGMENT STACK FOR ANTIALIASING Dec. 9, 1998

$m_{in} = m_{search} \wedge m_f$

$m_{out} = m_{search} \wedge \sim m_f$

And, the interesting thing that we can do, is to change a recursive calculation to a purely iterative one by knowing that compositing can be made fully associative, if computed with transparencies. The accumulated transparency is calculated in the shade/composite circuitry, and stored in the frame buffer. Essentially take the following equation from Carpenter $C = C_{in} \times A_{in} + C_{out} \times (1 - A_{in})$, as recursed for our example

$$C = C_{in}A \times A_{in}A + C_{out}(1 - A_{in}A)$$

$$C = C_{in}B \times A_{in}B + C_{out}(1 - A_{in}B)$$

$$C = C_{in}C \times A_{in}C + C_{out}(1 - A_{in}C)$$

$$C = C_{in}D \times A_{in}D + \emptyset$$

And convert it to purely a sum in terms of Transparency:

$$C = C_{in}A \times A_{in}A + \tau_A C_{in}B \times A_{in}B + \tau_{A-B} C_{in}C A_{in}C + \tau_{A-C} C_{in}D \times A_{in}D$$

Four separate contributions from the four separate layers $\tau_A = (1 - A_{in}A)$.

Craig Witterbrink Dec 9, 1998 To be continued

Craig Witterbrink

Thomas Madsen 12/14/98 Michael E. Goss 12/15/98

CONT: FRAGMENT STACK FOR ANTIALIASING Dec 10, 1998

The equations on the previous page show how the recursion may be converted to direct evaluation. Fig 4.4-1 shows how the masks for coverage of each fragment is initially calculated. The A_{in} in the compositing equations are those masks, constrained by the search mask, as the sorting processes front-to-back, a search mask for the pixel is maintained and updated. As before fragments A, B, C & D are sorted, front-to-back.

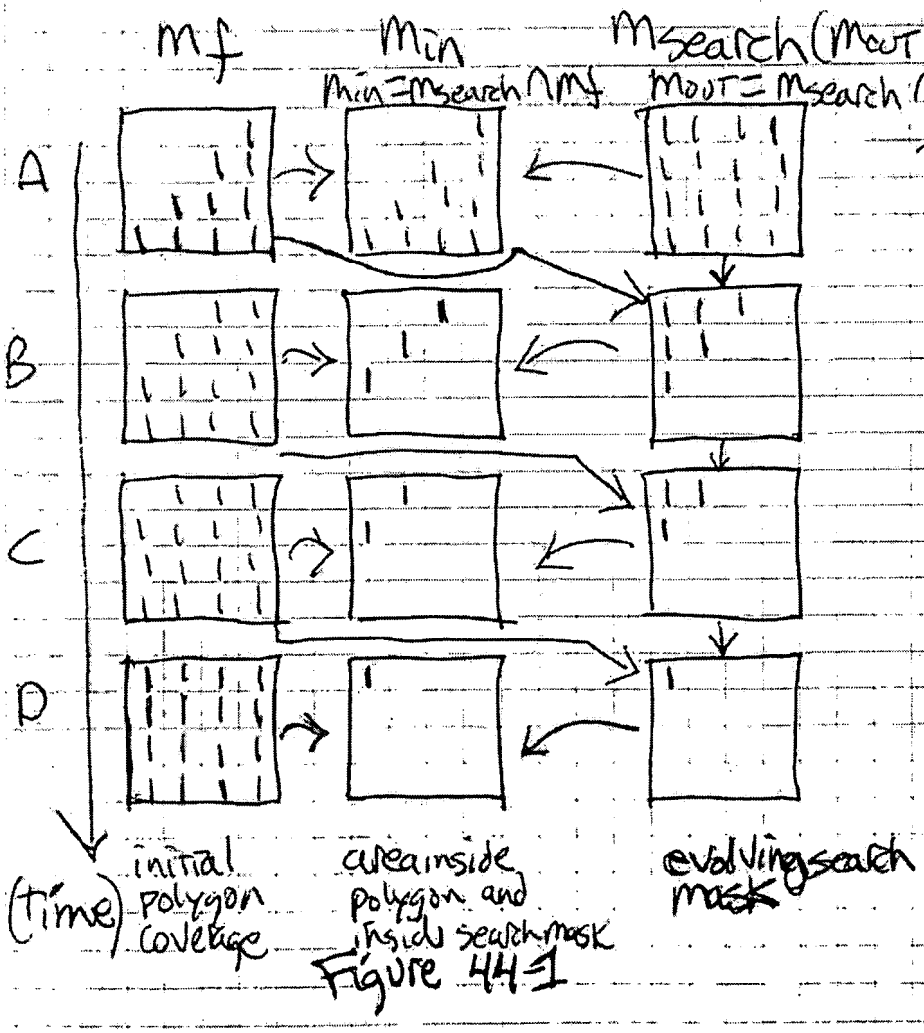


Figure 44-1

The search mask is updated front-to-back, and used at each step to compute the A_{in}, or as an approximation to the area. The 2nd column $m_{in} = m_{search} \cap m_f$ is used for A_{in} at those steps. And each A_{in} is used to update the accumulating transparency.

A_{in} for A = m_{in} for A, 10/16. Then $\tau_A = 1 - 10/16$.
A_{in} for B = m_{in} B = 3/16
$\tau_B = 1 - 3/16$, and

$$\tau_{A+B} = (1 - 10/16)(1 - 3/16)$$

$$\tau_{A+C} = (4/16)(13/16)(14/16)$$

Continued on Page 45

Read and Understood By

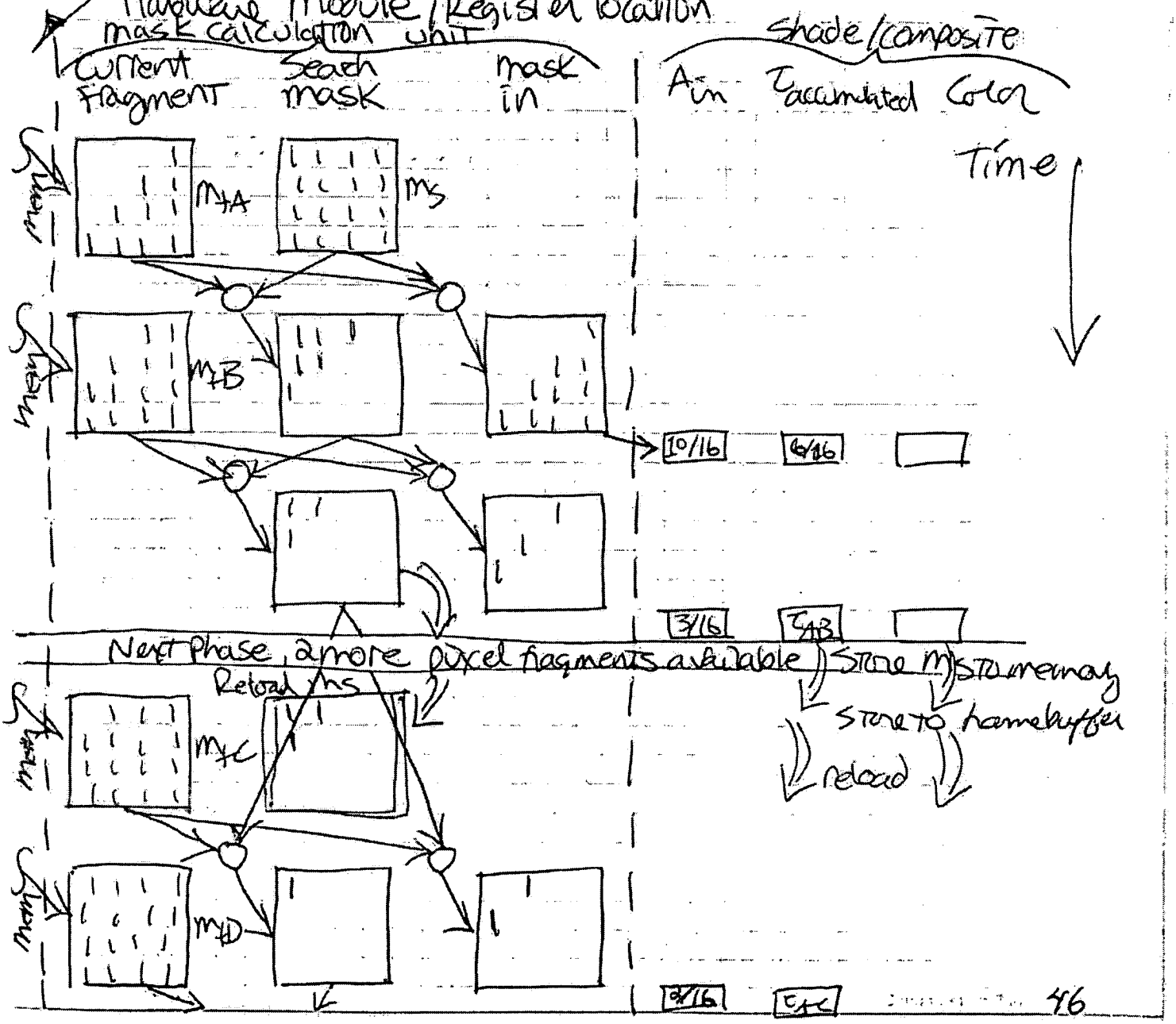
Thomas Malloy 12/14/98
Signed Date

Michael C. Grossman 12/15/98
Signed Date

CONT. FRAGMENT STACK FOR ANTI ALIASING

Dec 10, 1998 Cont

So, to simply show how the equation on page 43 is built up in Time, I show below the pipeline processing and calculating for the final result. The current fragment and the search mask are used to compute mask in and the next search mask, (3 Memories 2 buffers)



46

Thomas Malzbender 12/14/98 Signed Date

Michael E. Goss 12/15/98 Signed

CONT, FRAGMENT STACK FOR ANTIALIASING Dec 14, 1998 cont.

The search mask is computed by $M_S = M_{OUT} = M_S \cap \sim M_{f(current)}$ and the mask for inside the fragment is computed by fragment

$M_{\cap}$ current fragment $= M_S \cap M_{f\,current\,fragment}$.

The mask calculation unit has space for 3 masks. The current fragment mask is loaded from one of the attribute buffers, A_1 or A_2. The masks are used to compute the opacity or roughly the area by summing the bits in the min-mask.

When the fragment pixel location has been processed, the search mask for that location must be stored to memory into the attribute buffer, and reloaded. The same is true for the Transparency, and the accumulated color.

Note that one row of processing is missing, that to compute M_{IND}. To implement A-buffer, the search mask would have to be added to random access memory storage as in Z_1, A_1, Z_2, A_2, M_S for the sort and compare memories.

Craig M. Wittenbrink

Craig Wittenbrink

Dec 14, 1998.

As an additional generalization, it is instructive to point out how the fragment stack can enhance functionality of a conventional Z-buffer implementation. What you can implement is correct transparency, with more passes.

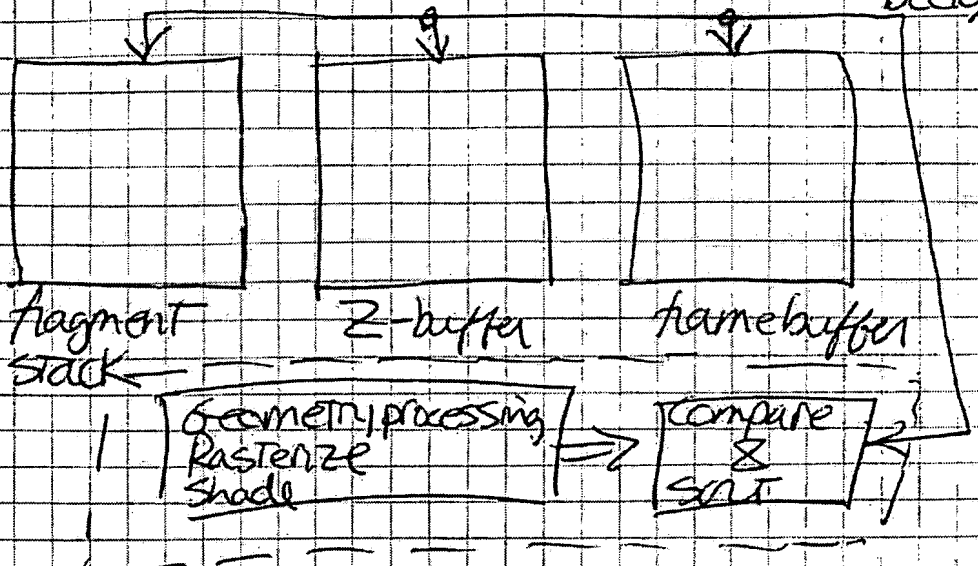
Continued on Page

Read and Understood By

Thom Malsbenden — Signed — 12/14/98 — Date

Michael E. Goss — Signed — 12/15/98 — Date

Dec 10, 1998 cont.



So, the conceptual sketch is provide the compare and sort hardware access to the fragment stack, the Z-buffer, and the name-buffer. The fragment stack is sized to support the typical depth complexity. When it's the stack overflows, it may be paged to main memory, because it is only accessed on the next phase, and the order of access is unimportant. Victimization strategies may be used, or stalling can be used to start compositing layers until enough storage is used up. Such victimization would affect performance and/or quality if a second Z-buffering was not done.

Key computational hardware required is the ability to change the comparison criteria to do back-to-front, front-to-back, even mask updates. This invention is general enough to improve a great number of architectures with little application impact.

The control loops for the fragment stack with a conventional Z-buffer are

Next Page

Continued on Page 48

Read and Understood By

Signed	Date	Signed	Date
Thomas Malzbender	12/14/98	Michael C. Doss	12/15/98

Dec 10, 1993 cont.

phase 1 processing
if $Z < Z_{min}$ {
if opaque {
$Z_{min} = Z$, store Z in buffer + attribute/color
} else { /* if transparent */
send to stack
} /* $Z < Z_{min}$ */
else
discard occluded fragment

phase 2)
if ($Z < Z_{min}$) {
send to stack
} else discard occluded fragment

phase 3 and higher, Z buffer compare changes; go back-to-front

→ While fragment stack not empty
clear Z, Z_{min} (invalidate Z)
process stack, all fragments {
if $Z > Z_{max}$ { /* if Z not valid, place Z in and do not recirculate */
$Z_{max} = Z$
if (Z ~~not~~ valid)
send Z, and color to fragment stack
} else
send fragment to stack. /* fragment not furthest */
}
Now composite back-to-front pixels that have changed
└ Repeat phase 2 & higher

Algorithm for fragment stack with Z-buffer

For tile based hardware, there are additional improvements possible because of the reduced address to be stored with fragments on the stack. For example a 1600 x 1200 screen, perhaps 11 bits for x and y, for a 22 bit overhead per fragment. Say the average depth complexity is 5.

Continued on Page 49

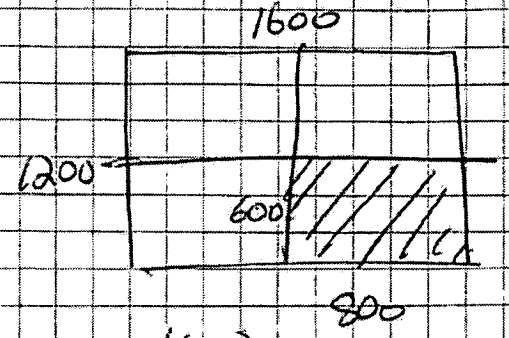
Read and Understood By

Thomas Malzbender — Signed — 12/14/98 — Date
Michael E. Goss — Signed — 12/15/98 — Date

Dec 19 1998 cont.

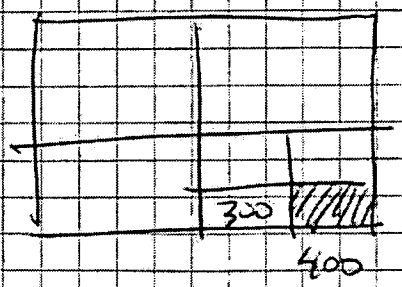
~~5760000~~

Take 4X1600X1200 fragments by 3 bytes of address data is ~~(83,886,080)~~ bytes. Tiling by 4 tiles To 800 x 600 tiles reduces the number of bits to 10 for X and Y. 2 bits for 21 bits encodes 2097152 addresses while 1600 x 1200 is 1,920000 addresses.



4 Tiles

1K X 1K Tiles



16 Tiles

512 X 512 Tiles

go to extreme, say 64X64 tiles, Requires 12 bits for X and Y, versus 22 bits, a 10 bit saving for fragments. A saving of roughly 7.6 million bytes of memory, if they were dedicated 64X64 tiles, for an average 5 depth complexity. The depth complexity of 5 assumes Transparent layers are not occluded. For most graphics the depth complexity is considerably less, so less memory would be consumed. If primitives were bucketized, and there was only a 64X64 tile of processing, The fragment buffer would be 24,576 bytes, a tiny fraction of memory, making the approach even more practical for tiling architectures, like Talisman [Torborg and Kajiya]

[Torborg and Kajiya] Jay Torborg and James T. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC", In Proceedings of SIGGRAPH, 1996, pages 353-363, New Orleans, LA.

The examples of two dedicated Z-buffers, and of using a traditional Z-buffer show how to generalize the idea to N buffers Craig M. Wittenbrink Craig Kolb Reed

Continued on Page

Read and Understood By

Thomas Malzbender 12/14/98 Signed Date

Michael E. Goss 12/15/98 Signed Date

graphics

Sep 9, 1999

Fragment buffer with a conventional Z buffer.

In order to maximize use of framebuffer memory as simple change to the invention on page 2373/47 to 2373/49 is to also store transparent fragments in the frame buffer. This requires storing an α, opacity, or alpha buffer as well but for scenes dominated by transparent polygons this is a big win.

The phase testing will change slightly. Now both transparent and opaque fragments will be placed in the buffer. The comparison test will vary depending on 1) cleared/uninitialized 2) transparent and 3) opaque fragments.

Sep. 15, 1999

many configurations are possible, with tradeoffs in memory versus time. The current range of scenarios include 1) regular Z-buffer, NO added framebuffer memory 2) A Z-buffer with double the Z's but only a single attribute storage 3) double Z and attribute storage, 4) more storage than that.

In all cases a single fragment buffer for screen region would be found. The number of passes would vary though.

Quick calculations suggest that the numbers passes made over the fragment buffer are

	# of passes	Example
case 1) regular Z-buffer	$2n+1-1$	9
case 2) add additional frame of Z storage	$n+1$	6
case 3) double buffer, add addtl. frame of att. storage	$n-1$	4
case 4) fragment stuck per pixel	$\lceil n/F \rceil$	3

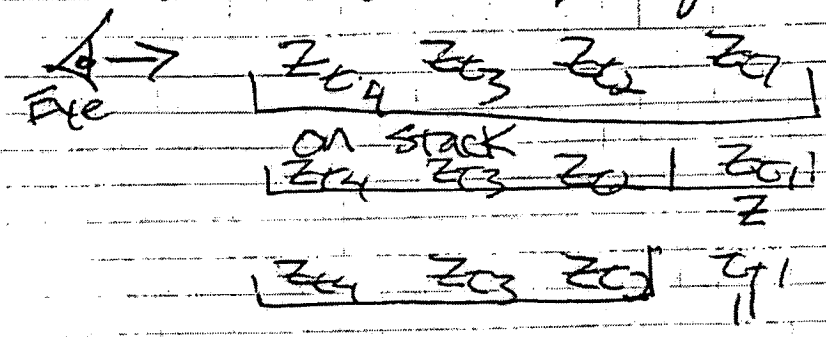
F = # of fragment attributes buffered

As an example consider the highest depth complexity for any pixel to be 5 with the following fragments, and scenarios for each of the 4 memory cases

Kevin Wu Nov. 9, 1999

Fragment buffer cont. 9-15-99 cont.

Case 1 No additional storage



on stack

Zc4 Zc3 Zc2 | Zc1 Oz Phase 2
z
occlude any transparent behind Oz - opaque z
Save Z of furthest transparent

Zc4 Zc3 Zc2 Zc1
||

Phase 2 + match z of furthest Transparent and blend

Phase 3 find next further z

Zc4 Zc3 | Zc2
z

phase 3+ match z and blend

Zc4 Zc3 | Zc2
||

Phase 3 find next further z

Zc4 | Zc3
||
Zc4 | Zc3

Phase 3+ match & blend

Phase 3 find next further

Zc4
||
Zc4

Phase 3+ match & blend

9 passes, or $2n+1$. n = depth complexity of most complex pixel. So, as can be seen, the fragment buffer with a single z buffer does a lot of work to compute correct transparency. For low depth complexity or low average depth complexity this is acceptable.

Continued on Page 65

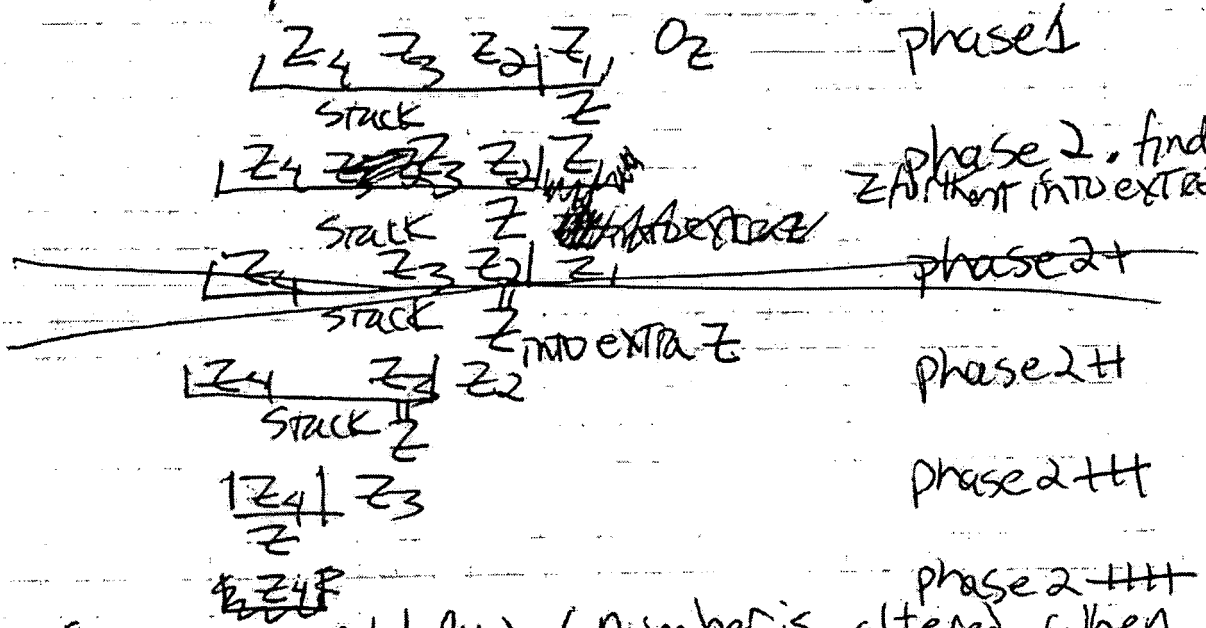
Read and Understood By

Kevin Wu Nov. 9, 1999
Signed Date Signed Date

Fragment buffer cont. 9-15-99 cont

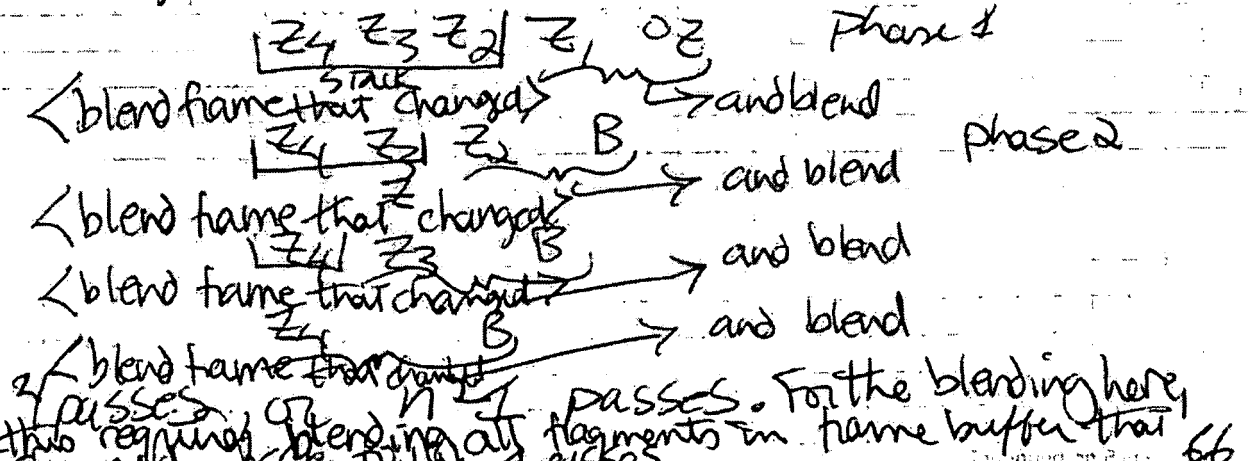
Now if an additional frame buffer of ~~Z~~storage is added the complexity may be reduced by a factor of 2.

Case 2, add additional frame of Z storage



6 ~~passes~~ n+1 CW passes, (number is altered when considering fragments stacked beyond page)

Case 3, double buffer Z buffers and 2 attribute buffers processing is similar to above

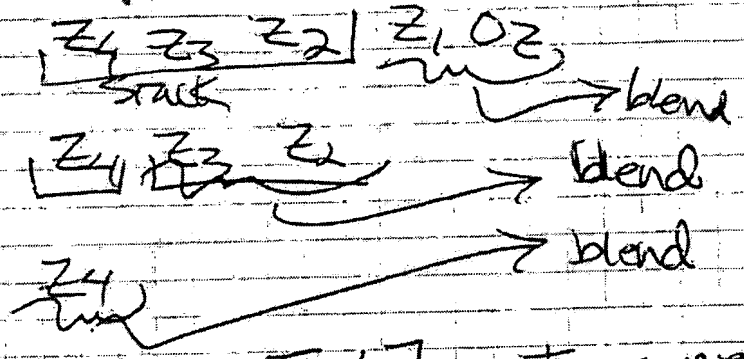


2 passes ~~or~~ n+1 passes. For the blending here, two requiring blending all fragments in frame buffer that changed, essentially 3 passes 66

Kevin Wu Nov. 9, 1999

Fragment buffer 9-15-99 cont?

case 4 is the separate Z and attribute buffers from the frame buffer, stake Two as an example



3 passes, or $\lceil n/2 \rceil$. In general for k buffers there would be $\lceil n/k \rceil$ passes.

The algorithm pseudo code description for the fragment stack processing, only with a regular Z buffer is

Conventional 1 pass

phase 1 determine closest opaque else furthest transparent. Note that some transparent fragments may get into buffer that should be culled.

2 passes

phase 2 Blend furthest transparent, else will behind opaque
→ save Z of furthest transparent for transparent only pixels, but don't update frame colors, just Z.
→ then make pass looking for fragments that match and blend them. In areas with opaque no blending will occur

2 passes per additional layer

phase 3
→ find next furthest Z
→ then make pass to match and blend them
→ repeat until no additional fragments

Algorithm for case 1, regular Z buffer

Continued on Page

Read and Understood By

Kevin Wu — Signed
Nov. 9, 1999 — Date
Signed — Date

9-15-99 cont.

Fragment buffer cont.

Comments on Z buffer only, no alpha buffer is needed for back TO front. But to do specific weighted compositing you would need an alpha buffer for the best precision.

The case 3 and 4 require post processing of the frame buffer to determine ~~pixels~~ that fragments that are updated to blend them. While scheme (case 1 & 2) you just blend fragments that match.

10/7/99

To determine the specific controls necessary for a complete fragment buffer implementation, the most promising case was selected. The asymptotic complexity with depth complexity per pixel of n is: (keeping multiplicative constants for comparison)

case1	O(2n)
case2	O(n)
case3	O(n)
case4	O(n/4)

Case 2 provides the first big jump in improvement for nominal cost, an extra Z buffer, beyond case 1. Therefore case 2 is explored in detail in the following.

The scenario is as follows, the architecture is a standard graphics pipeline with geometry processing (G) and rasterization (R). We add a fragment stack, and Z storage to the frame buffer.

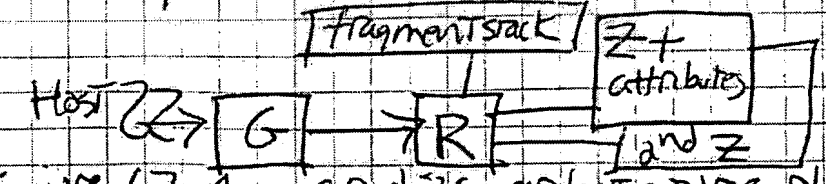


Figure 67-1 graphics architecture, plus added fragment stack and 2nd Z buffer.

Continued on Page 68

Read and Understood By

Signed	Date	Signed	Date
Kevin Wu	Nov. 9, 1999		

Fragment Buffer Cont'd 10/7/99 cont.

More details of how the fragment stack and frame buffer are accessed are given below:

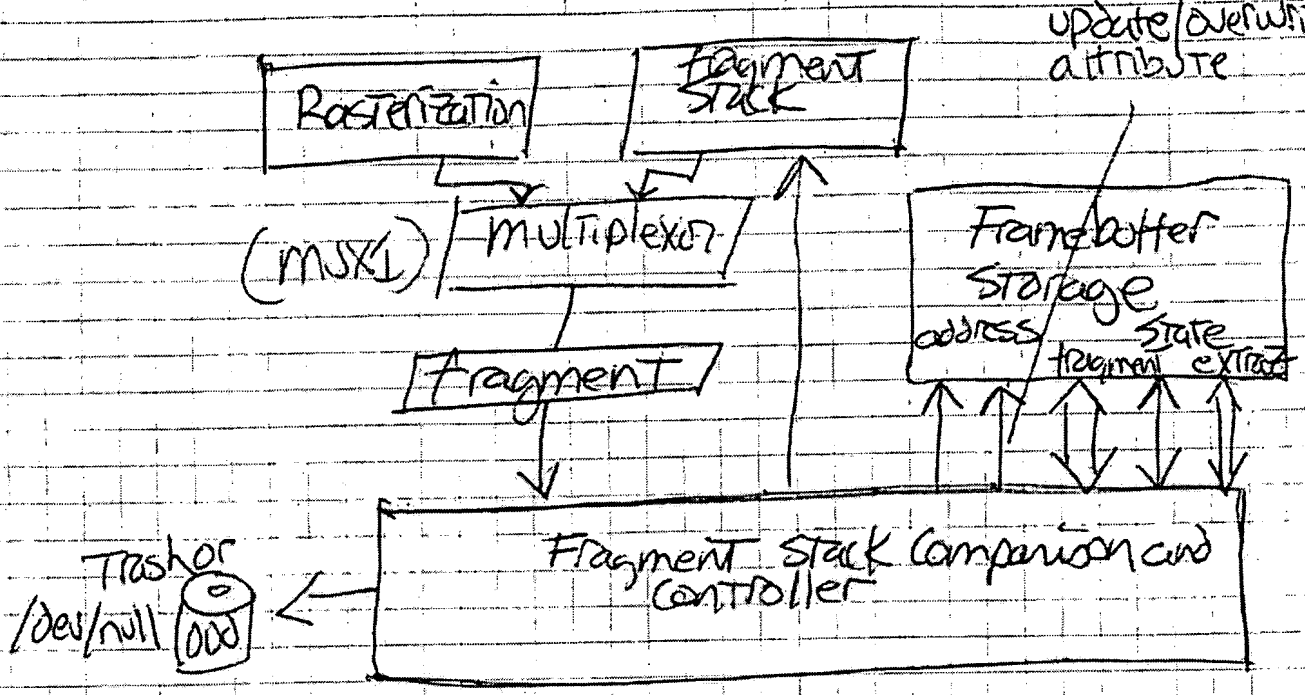


Figure 68-1 Detailed diagram of interface to fragment buffer and frame buffer.

Figure 68-1 shows many details used for one particular implementation of the fragment buffer. Shown is a multiplexor that can choose the input either as the Rasterization unit, or the fragment stack (MUX1). The fragment is fed into the fragment stack comparison and controller. This is where the Z-buffering comparison typically takes place. The Z-buffering is now augmented, revised to provide true transparency and/or anti-aliasing as discussed previously. The case 2 example is going to be presented in more detail, and then one implementation of the fragment buffer will be shown.

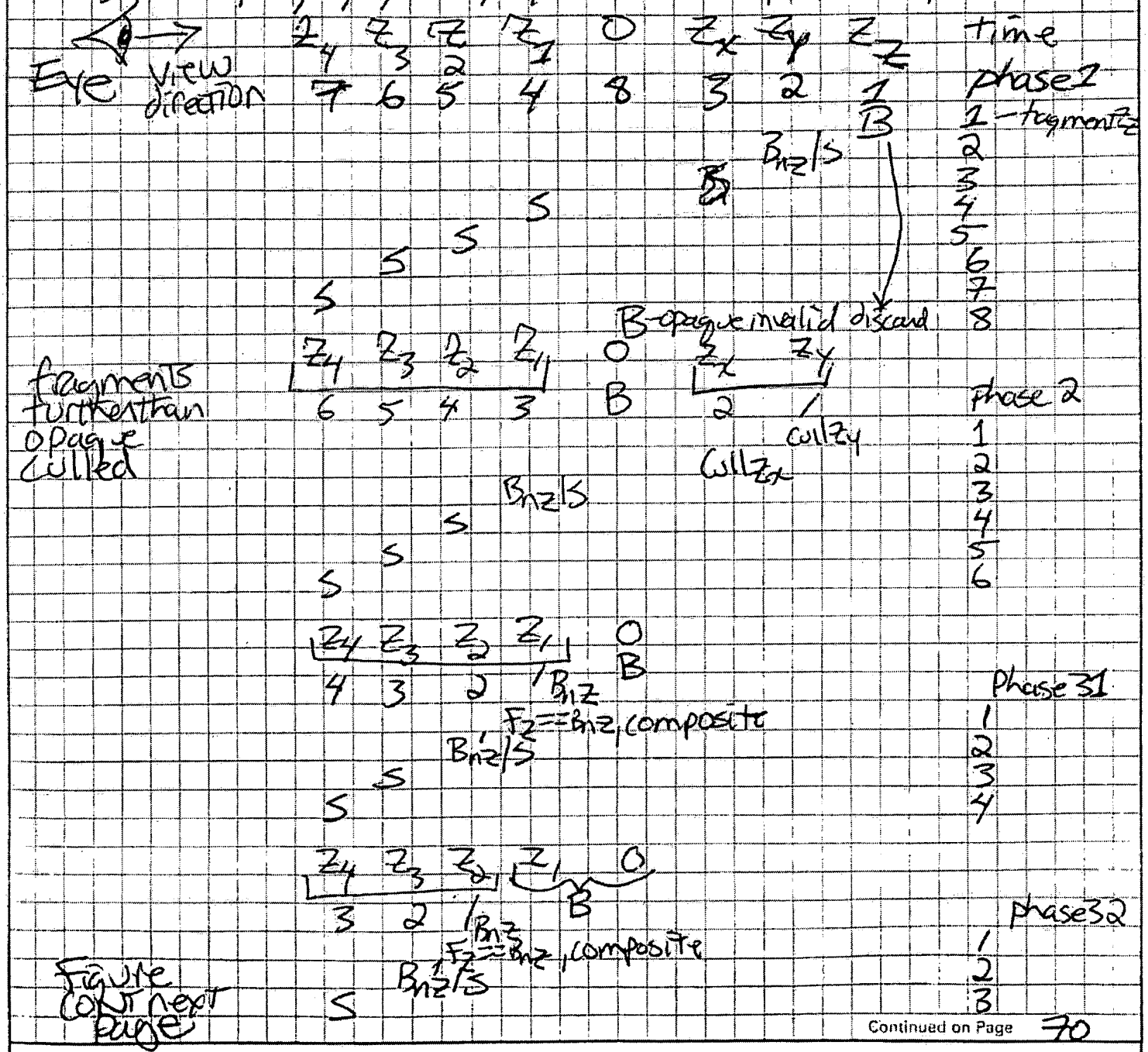
Continued on Page 69

Read and Understood By

Kevin Wu — Signed
Nov. 9, 1999 — Date
Signed — Date

Fragment buffer cont. 10/7/99 cont.

Given the following pixel with 8 fragments, one opaque fragment 1, 0, and drawn in the order shown, 1, 2, 3, ..., 8, Edse-2 processing occurs.



Continued on Page 70

Read and Understood By

Signed Date Signed Date

Kevin Wu Nov. 9, 1999

Fragment Buffer cont.

Oct 7, 1999

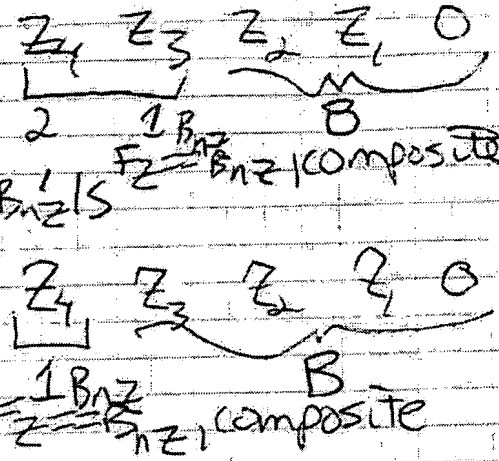


Figure 69-70-1, Case 2 fragment processing example.

Figure 69-70-1 shows how during phase 1, any opaque layers are found. In this case, transparent fragments beyond the opaque layer were stacked. In phase 2, these fragments are culled, and the furthest transparent layers Z3 saved. The fragments are processed in the same order each time as they are read from and written to the stack. Note that the depth complexity of the pixel is 5, but we took 6 passes. In cases where no further than opaque → transparent → fragments are stacked there is one less pass, 5. Phase 2 culls fragments Z2 and Z4 as they are further from the eye point than the opaque layer 0. Then in phase 2 fragment Z1 has its Z value saved as BNZ, and is put on the stack, as shown by 5. Then fragments Z2, Z3, & Z4 are also re-stacked. In phase 31 the frame buffer B, holds the opaque Z, and color attributes. BNZ or BNEXTZ is also in the frame buffer, and holds the proper Z value for the furthest transparent layer.

Continued on Page 71

Read and Understood By

Kevin Wu — Signed
Nov. 9, 1999 — Date
Signed — Date

Fragment Buffer cont. OCT 8, 1999

Cont, phase 31, The fragments come out of the fragment buffer in the same order that they were placed there. FIRST, fragment Z_4 is read, and it's Z value matches BNZ. Therefore, it is immediately composited with the frame buffer, using for example the Porter and Duff Over operator. The next fragment is read, Z_2, and it is the furthest Z that is closer than the BNZ, so BNZ (BNext Z prime) is written, and the fragment is re-stacked (S). Fragments Z_3 and Z_4 are considered and re-stacked (S).

Phase 32 continues the same as phase 31, with the remaining fragments on the stack. Here, are Z_2, Z_3, and Z_4. Note that the BNZ on BNext Z prime of phase 31 is BNZ, or BNext Z of phase 32. This alternation of the interpretation of BNZ and BNZ' continues for each even and odd phase 3 needed. The Z storage used is the frame buffer Z and the 2nd frame buffer Z. Always Rust 2 Z values per pixel.

physical location	phase1	phase2	phase3 odd	phase3 even	phase3 odd
BNZ	BNZ	BNZ	BNZ'	BNZ	BNZ'
BZ	BZ	BNZ'	BNZ	BNZ'	BNZ

The phase next Z prime (BNZ') and BNext Z (BNZ) alternate interpretation, while their location is in BNZ or BZ of the framebuffer physical location, see I on the left.

Phase 33 starts with a Buffer, B, that is the previously composited opaque, and furthest two transparent fragments. BNZ is the Z value of fragment Z_3. Z_3 is the first fragment considered, so it is matched with BNZ and also composited. Fragment Z_4 is set BNZ'. Then in phase 34, fragment Z_4 is considered again, matches BNZ, and is composited into the final correct pixel color.

The state machine was designed Sep 21, 1999, and is reproduced here.

Continued on Page 72

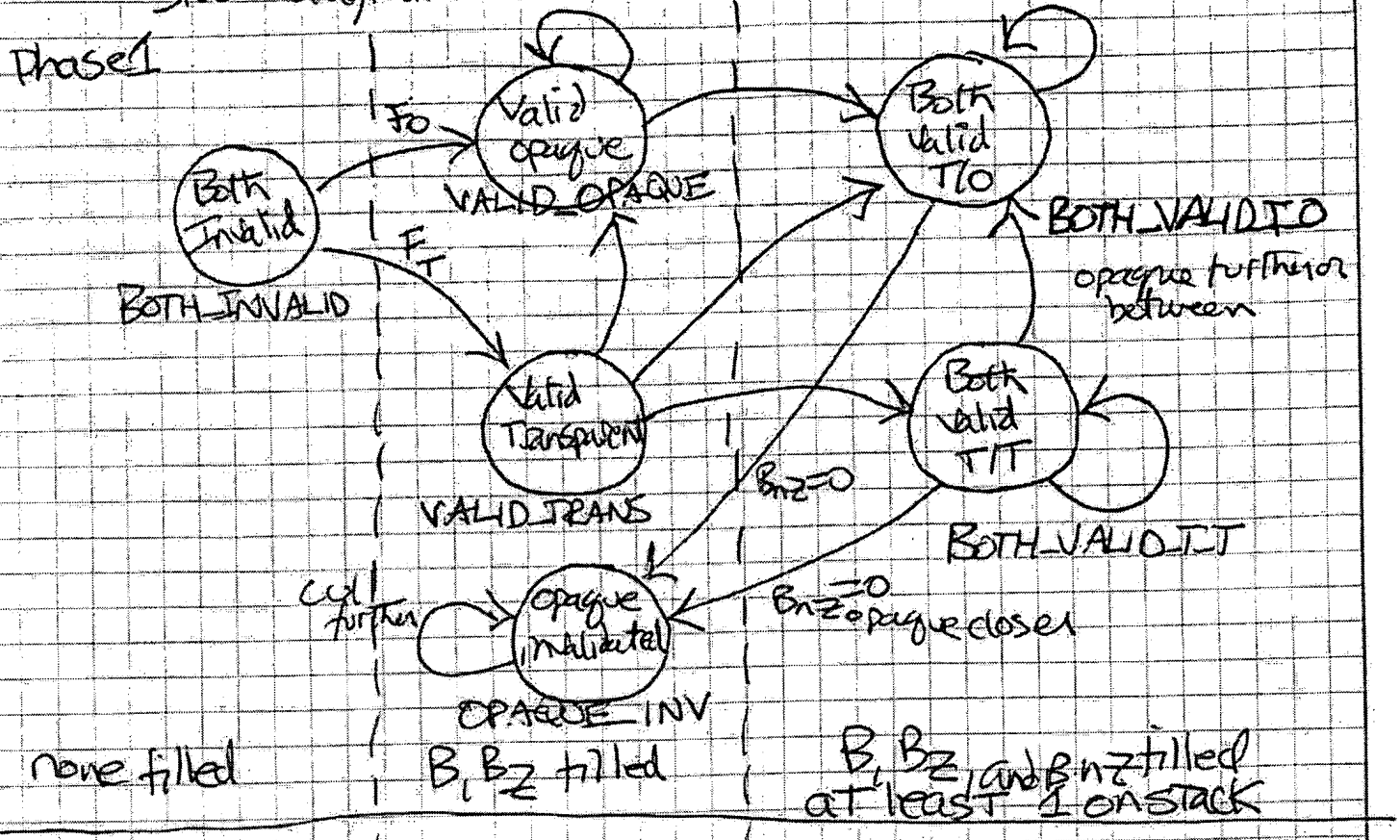
Read and Understood By

Kevin Wu — Nov. 9, 1999 — Signed — Date — Signed — Date

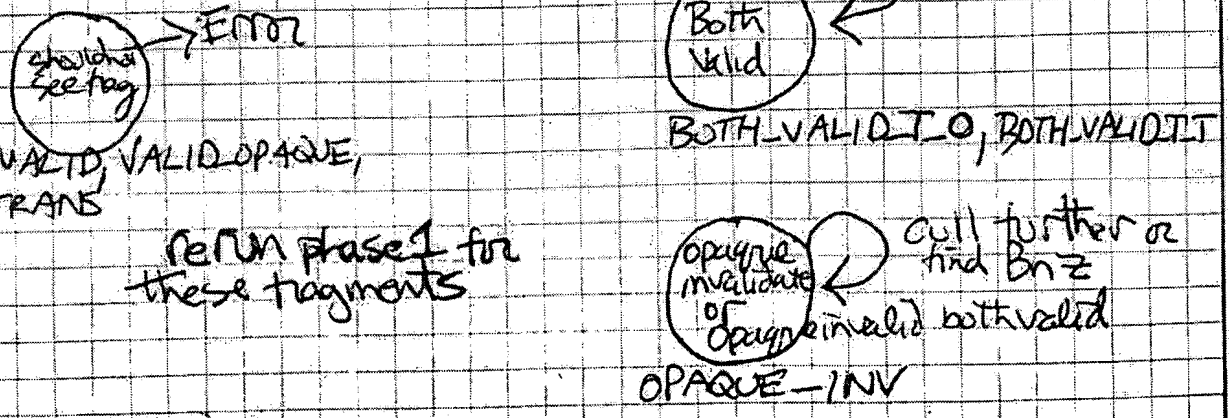
Fragment Buffer cont.

Oct. 8, 1999

State Diagram for Phase control



phase 2

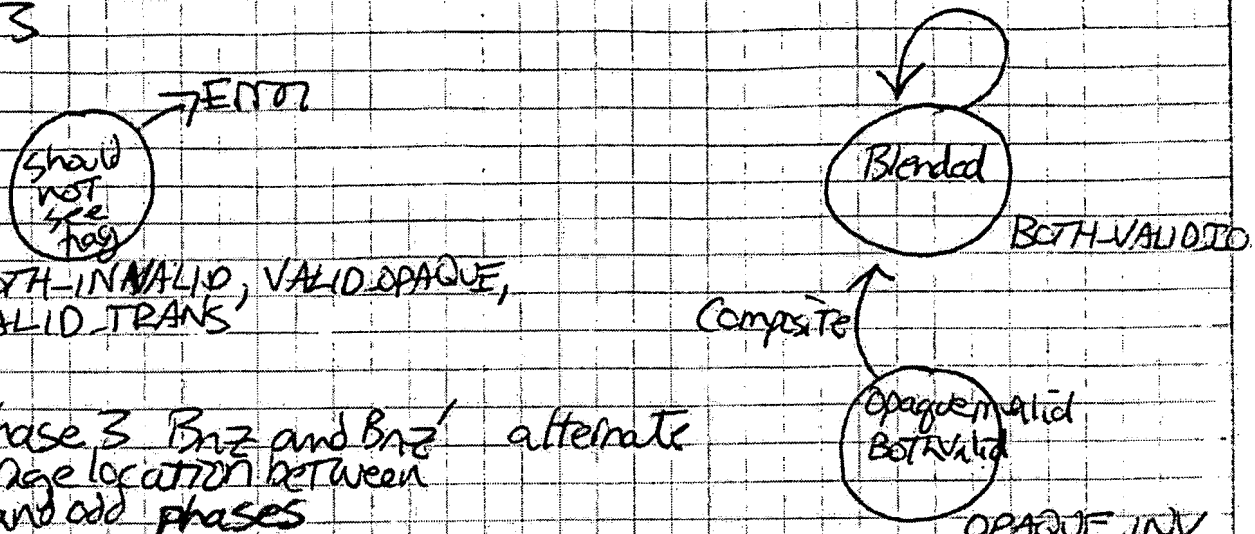


Continued on Page

Read and Understood By

Kevin Wu — Signed
Nov. 9, 1999 — Date
Signed
Date

phase 3



for Phase 3 Bnz and Bnz' alternate in storage location between even and odd phases

Figure 72-73-1, phase 1, phase 2, & phase 3 state diagrams.

Each framebuffer location has a unique state. So each pixel pa state machine only, you just need to consider the state machine for the fragment location. The state machine require 6 states for phase 1, 3 states for phase 2, and 3 states for phase 3. The 6 states have been labelled as

BOTH_INVALID – initial state, no fragment seen here
VALID_OPAQUE – 1 opaque fragment seen & stored
VALID_TRANS – 1 transparent fragment seen & stored
OPAQUE_INV – an opaque fragment stored closer than stored
BOTH_VALID_T_O – at least TWO fragments, opaque & trans.
BOTH_VALID_T_T – at least two fragments both transparent

The same state assignments are used for all 3 phases. Of course interpretation varies between phases. The 2nd and 3rd phases will generate an error, if a fragment is located, where zero or 1 were seen in phase 1. phase 1 states have been separated into 3 columns, from the left, no fragment framebuffer filled, or BOTH invalid, middle, B and Bz filled, and right B, Bz and Bnz filled and at least 1 fragment on the STACK.

Continued on Page 74

Read and Understood By

Kevin Wu — Signed
Nov. 9, 1990 — Date
Signed
Date

Oct 8, 1999

phase1 Fragment Buffer cont'd

Current State	INPUTS	OUTPUTS/Side effects	Next State
BOTH_INVALID	F_O	B=F	VALID_OPAQUE
BOTH_INVALID	F_T	B=F	VALID_TRANS
VALID_OPAQUE	$F_Z \geq B_Z$	none (call fragment)	VALID_OPAQUE
VALID_OPAQUE	F_O, $F_Z \leq B_Z$	B=F	VALID_OPAQUE
VALID_OPAQUE	F_T, $F_Z < B_Z$	$B_{NZ}=F_Z$, stack(F)	BOTH_VALID_T_O
VALID_TRANS	F_O, $F_Z \leq B_Z$	B=F (call B replace frame)	VALID_OPAQUE
VALID_TRANS	F_O, $F_Z > B_Z$	stack(B), $B_{NZ}=B_Z$, B=F	BOTH_VALID_T_O
VALID_TRANS	F_T, $F_Z \leq B_Z$	$B_{NZ}=F_Z$, stack(F)	BOTH_VALID_T_T
VALID_TRANS	F_T, $F_Z > B_Z$	stack(B), $B_{NZ}=B_Z$, B=F	BOTH_VALID_T_T
BOTH_VALID_T_O	(F_X), $F_Z \geq B_Z$	none (call fragment)	BOTH_VALID_T_O
BOTH_VALID_T_O	F_O, $F_Z < B_Z$ & $F_Z > B_{NZ}$	B=F	BOTH_VALID_T_O
BOTH_VALID_T_O	F_O, $F_Z < B_Z$ & $F_Z \leq B_{NZ}$	B=F, $B_{NZ}=0$ (read B_{NZ})	OPAQUE_INV
BOTH_VALID_T_O	F_T, $F_Z \leq B_Z$ & $F_Z > B_{NZ}$	$B_{NZ}=F_Z$, stack(F)	BOTH_VALID_T_O
BOTH_VALID_T_O	F_T, $F_Z < B_Z$ & $F_Z \leq B_{NZ}$	stack(F)	BOTH_VALID_T_O
BOTH_VALID_T_T	F_O, $F_Z > B_Z$	$B_{NZ}=B_Z$, stack(B), B=F	BOTH_VALID_T_O
BOTH_VALID_T_T	F_O, $F_Z \leq B_Z$ & $F_Z > B_{NZ}$	B=F, (replace frame)	BOTH_VALID_T_O
BOTH_VALID_T_T	F_O, $F_Z \leq B_Z$ & $F_Z \leq B_{NZ}$	B=F, $B_{NZ}=0$ (replace frame)	OPAQUE_INV
BOTH_VALID_T_T	F_T, $F_Z > B_Z$	$B_{NZ}=B_Z$, stack(B), B=F	BOTH_VALID_T_T
BOTH_VALID_T_T	F_T, $F_Z \leq B_Z$ & $F_Z > B_{NZ}$	$B_{NZ}=F_Z$, stack(F)	BOTH_VALID_T_T
BOTH_VALID_T_T	F_T, $F_Z \leq B_Z$ & $F_Z \leq B_{NZ}$	stack(F)	BOTH_VALID_T_T
OPAQUE_INV	$F_Z \geq B_Z$	none (call fragment)	OPAQUE_INV
OPAQUE_INV	F_O, $F_Z < B_Z$	B=F	OPAQUE_INV
OPAQUE_INV	F_T, $F_Z < B_Z$	stack(F)	OPAQUE_INV

State transition table for phase 1. stack(X) means To place fragment X on fragment buffer or stack.
B- frame buffer, F- fragment being considered
F_O- fragment opaque, F_T- fragment transparent
F_Z- fragment depth value B_Z frame buffer depth value
B_{NZ}- frame buffer next Z depth value

Continued on Page 75

Read and Understood By

Kevin Wu — Nov. 9, 1999 — Signed — Date — Signed — Date

Hewlett-Packard Laboratories

PROJECT

Continued From Page

Fragment Buffer cont.

phase 2

Current State	inputs	outputs (side effects)	Next State
BOTH_VALID_I_O	$F_Z == B_{nZ}$	B = Composite(B, F)	BOTH_VALID_I_O
BOTH_VALID_I_O	$F_Z != B_{nZ}, B_{nZ}' > B_{nZ}$	$B_{nZ}' = F_Z$, STACK(F)	BOTH_VALID_I_O
BOTH_VALID_I_O	$F_Z != B_{nZ}, B_{nZ}' < B_{nZ}, F_Z \leq B_{nZ}'$	STACK(F)	BOTH_VALID_I_O
BOTH_VALID_I_O	$F_Z != B_{nZ}, B_{nZ}' < B_{nZ}, F_Z > B_{nZ}'$	$B_{nZ}' = F_Z$, STACK(F)	BOTH_VALID_I_O
OPAQUE_INV	(F_), $F_Z > B_Z$	none (will fragment)	OPAQUE_INV
OPAQUE_INV	(F_), $F_Z < B_Z$ & $F_Z > B_{nZ}$	$B_{nZ} = F_Z$, STACK(F)	OPAQUE_INV
OPAQUE_INV	(F_), $F_Z < B_Z$ & $F_Z \leq B_{nZ}$	STACK(F)	OPAQUE_INV
(provides same behavior as BOTH_VALID_I_O phase 1)			

phase 3

Current State	inputs	outputs (side effects)	Next State
BOTH_VALID_I_O — same as phase 2 above	— above —	— above —	— above —
OPAQUE_INV ~~OPAQUE_INV~~	$F_Z == B_{nZ}$	B = Composite(B, F), $B_Z = B_{nZ}$ (or $B_{nZ}' = B_{nZ}$) $B_{nZ} = F_Z$ already	BOTH_VALID_I_O
OPAQUE_INV	$F_Z != B_{nZ}$	$B_Z = F_Z$, STACK(F) (or $B_{nZ}' = F_Z$)	BOTH_VALID_I_O

OPAQUE_INV only occurs in phase 31, not in phase 32, phase 33, etc.

For phase 3, the B_{nZ} and B_{nZ}' are in different locations depending on even or odd. Phase 3 looks like an even phase 3

Physical location	phase 1	phase 2	phase 31 odd	phase 32 even ...
BNZ	BNZ	BNZ'	BNZ	BNZ'
BZ	BNZ'	BNZ	BNZ'	BNZ

Continued on Page 76

Read and Understood By

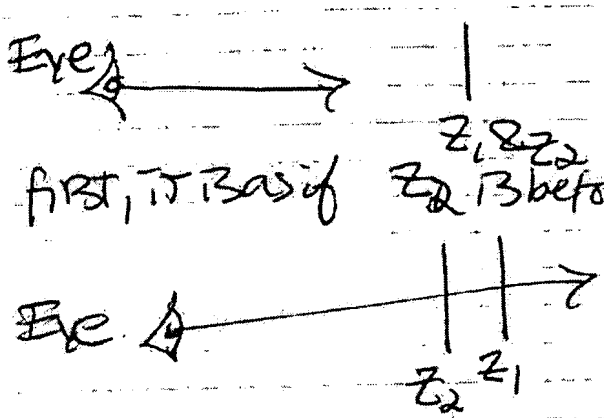
Kevin Wu — Nov. 9, 1999 — Signed — Date

Signed — Date

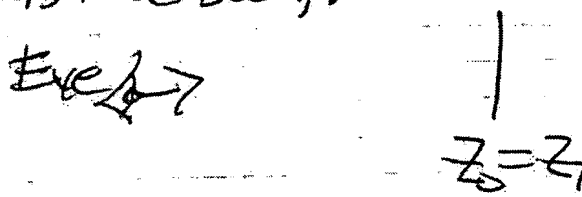
fragment buffer cont.

Oct 8, 1999

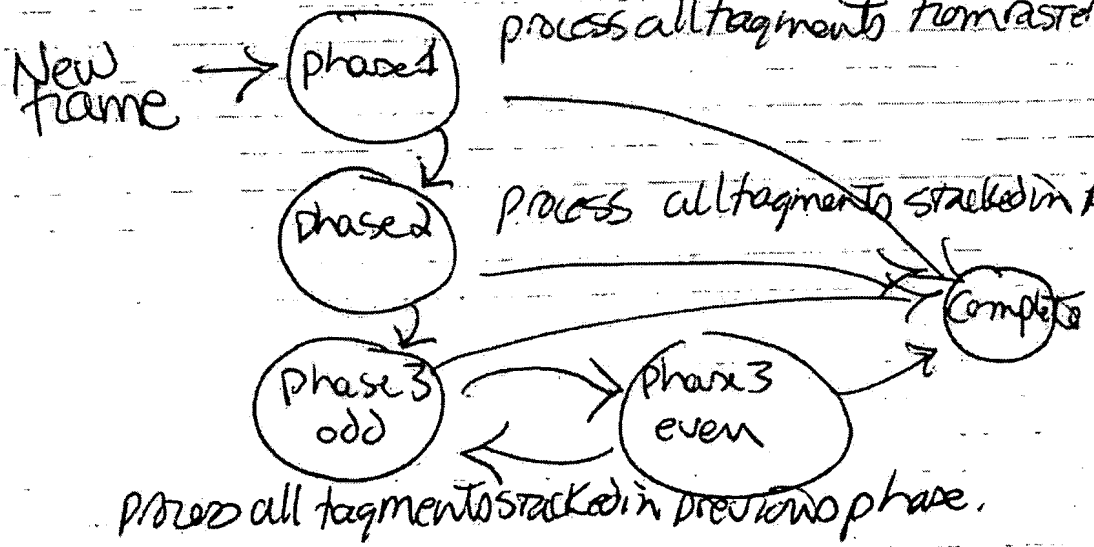
For this implementation the rules for equal valued depths are that the earlier fragment is in back of the following fragments



For interaction with opaque, if a transparent fragment is to be seen, it must be less than the opaque z



B: The overall state machine, that controls evaluation



Kevin Wu Nov. 9, 1999

fragment buffer.mtl

OCT 3, 1999

After the fragment stack is emptied, the rendering and compositing is complete. If there are no more than 1 visible (opaque or transparent) fragment per pixel, then processing completes in phase 1. With only 2 fragments in a pixel, that are not occluded, processing may complete in phase 2, and so on.

This state machine has been fully implemented and debugged. Several complex models work run through it.

Craig M. Wittenbrink

Craig M. Wittenbrink

OCT 3, 1999

Continued on Page

Read and Understood By

Kevin Wu — Nov. 9, 1999

Signed — Date — Signed — Date